Chapter 1

Using FDR

FDR is, at the time of writing, the most important and useful tool for the automatic analysis of systems described in CSP. Indeed it is it that makes CSP such a practical tool for the specification, analysis and verification of systems.

In this chapter we will describe the main functionality of FDR and guide the reader in how to use it effectively. It is intended that it is studied in parallel with the introductory section of this book: those studying CSP may well want to use FDR from day one, but will not be in a position to understand some of the basic functions of the tool until they have mastered failures and failures-divergence refinement (Chapter **??**).

The web site for this book contains an extensive library of CSP scripts, many of which are based on the examples in this book and [**?**]. To get to know FDR, the reader is recommended to try – and perhaps modify – some of these, and to implement his or her solutions to the exercises in this and other chapters. In addition to this, it contains a number of more extensive practical exercises, most of which have been used in courses at Oxford University.

The primary aim of the exercises in this chapter is to re-inforce understanding of how FDR works and how it interacts with CSP's theory.

1.1 What is FDR?

FDR can be described as a *refinement checker* or *model checker*[1] for CSP. It is a tool for checking properties of processes written in CSP and, when these are not

[1]A broad definition of a 'Model checker' is any tool which automatically establishes or refutes the proposition automatically that systems meet their specifications, by somehow exploring all relevant execution paths of the system, and exhibiting a counterexample behaviour if one if the states found does not meet the implied requirements. By a narrower definition, namely that the

true, giving details of one or more counter-examples.

It does not allow you to "run" CSP programs in any conventional way, though there is an associated CSP animator ProBE which allows you to explore how a program behaves. Rather, FDR allows you to make assertions about processes and then explores (if necessary) every possible behaviour of the target process to see if the assertions are true.

FDR operates by loading a script written in the language CSP$_M$, which is an ASCII syntax for CSP with the addition of a functional programming language. We have already seen the CSP$_M$ versions of the operators introduced in earlier chapters. All of the CSP processes, and most objects derived from them, are presented in CSP$_M$ in the present chapter, using `this typeface`.

Almost all CSP scripts define some processes, and FDR allows you to decide refinement relationships between these, and to check directly whether an individual process is deadlock-free, divergence-free or deterministic. These checks can be set up by choosing the appropriate mode of FDR and then selecting two (for refinement) or one (for the other checks) and sometimes a semantic model, or alternatively can be written into the input script. The screen shot, Figure **??**, shows the refinement mode on FDR 2.91, where the user has selected specification and implementation processes, and there are also several other, pre-loaded, checks displayed in the window below.

Aside from its refinement, deadlock, divergence and determinism modes, it will be apparent from the screen-shot that there is also evaluate. This is not for processes, but allows the user to calculate the value of any non-process expression, from simple expressions such as $2 + 2$ and N (where a value of N is defined in the script) to testing complex pieces of non-process programming present in the script.

An optional graph mode is also shown, which can be activated by ???. This allows the user to see a graphical representation of the states of a CSP process in the form that FDR explores. The graphs that are displayed would frequently be too large to understand if calculated for a full system implementation, so typically this mode is employed to help the user understand of one of the smaller (typically either the specification or a component) process he or she has created.

All of this is illustrated by the simple input script `intro_fdr.csp` which, like many other scripts associated with this book, can be down-loaded from the book's web site. This script has three sections of definitions and numerous `assert` lines, which pre-load checks. It was deliberately created without any of any parallel operator, so it can be understood by readers who have not yet reached Chapter **??**. In Section **??** we discuss how the use of parallel operators affects FDR.

specification is cast in some, usually temporal logic and that the role of the tool is to check that the implementation is a *model* of it, FDR is not a model checker.

It is important to understand how FDR represents processes. While almost every function it computes is based on one of CSP's semantic models, FDR never represents processes explicitly as sets of traces, failures, etc. This is just as well since for most practical systems these sets are infinite. Rather every process is represented by a labelled transition system (LTS): either of the sort described on page **??** or a variant on this theme. We will see more details of this representation and how FDR calculates it and then decides questions about it during this chapter.

1.1.1 Running and debugging checks

The usual type of refinement check that FDR performs is between a *specification* process `Spec` and an *implementation* process `Impl`: one asks FDR the question as to whether `Impl` refines `Spec` one of a number of models. In the first sections of this chapter we will mainly consider the *traces* model.

In the usual form of check, `Impl` is a process that we hope satisfies some specification represented by `Spec`, which might either be an extremely simple process or a more complex object such as a complete behavioural specification or an earlier stage in a process of development by successive refinement.

Later in this book we will meet a few cases where the process we are trying to prove a property of appears more than once on the right-hand side of a check, or even on the left hand side, but in this chapter we will concentrate mainly on straightforward checks of the form `Spec [X= Impl` and occasionally `Spec [X= Impl\Y`, where `Impl` is the implementation we are checking, `Spec` is a process that is defined independently of it, and `X` is the model over which refinement is calculated.

The first section of `fdr_intro.csp` declares three simple events and defines four processes.

```
channel a,b,c

P = a -> b -> P

Q = (a -> b -> Q) [] (c -> STOP)

R = (a -> b -> R) |~| (c -> STOP)

DIV = DIV |~| DIV
```

Many checks can be performed on FDR in examining and comparing these processes. `DIV` (which performs internal τ actions for ever) only has the empty trace `<>` and therefore trace-refines the other three. `P` trace refines `Q` and `R`, which are trace equivalent (i.e. refine each other). `P` is deterministic and free of deadlock and livelock. Both `Q` and `R` can deadlock (by reaching the `STOP` state) and are divergence free. `Q` is deterministic but `R` is not, since it can either accept or refuse either of `a` and `c` on the empty trace (see Section **??**). `DIV`, naturally, can diverge.

If you run an assertion will find that FDR prints a cross next to the check if it fails, and a tick if it succeeds. Double-clicking on a failed check will take you into the debugger, which shows you a behaviour of the implementation process that contradicts the refinement, or if appropriate finds a deadlock or finds a divergence. In the case of a determinism check that fails other than by finding a divergence, the debugger will show you two behaviours: one on which some event was accepted after a given trace, and one on which it was accepted after the same trace.

The great simplicity of the processes `P`, `Q`, `R` and `DIV` means that the debugger has nothing to analyse more deeply. We will see later that in more complex examples it can help you understand how a misbehaviour can be analysed to see how sub-processes behaved.

1.1.2 FDR's settings

In this section we discuss the settings in running checks on FDR, that are accessible through the range of buttons at the top of the GUI on FDR 2.91. (We anticipate a revised GUI for FDR 3, but most or all of the following options will still be there.)

The first button FILE gives you the options to load and re-load a script, to edit the current script, to run all the `asserts` in the current script, and to quit FDR.

The second button OPTIONS is the most interesting. The tabs on it are listed here in roughly the order of interest to a relatively new user.

- Show Status creates a separate window in which you can see output from FDR's various functions. You should normally select this to be open. The output you will see reports progress on the various functions of running a check: on simple checks you are likely to see output from the compiler and the main run of the check. It also displays both parsing and compiling errors for your script, and statistics on any run you have just completed.
- Messages controls the level of detail printed out during a check into the Status window. It defaults to a moderate amount (auto) in which the amount of detail gradually decreases as a check proceeds. Setting it to high can be very useful if you are having problems getting something to compile because it gives you great detail on the terms it goes though, and so in particular you will know where the problem was encountered.

- Restart is something to use if FDR gets into some sort of bad state and is not responding: it is a more decisive version of the INTERRUPT button, and makes no attempt to retain state from previous checks.
- Examples determines how many counter-examples FDR will look for when running a check. This defaults to 1 so that a check halts as soon as one counter-example is found. If a higher number is selected then the tool will try to find more counter-example states until the limit is reached. Two counter-examples are reckoned to be different here if their final states, namely the ones that exhibit a particular behaviour such as an event, refusal or immediate divergence that violates the specification, are different. It is possible that different counter-examples might have the same trace, and if there is only one counter-example state with lots of traces leading to it, only one trace will be reported. For example, if you ask for more than one example on the following example:

  ```
  P = a -> c -> STOP
     [] a -> a-> c -> STOP
     [] a -> c -> c -> STOP

  Q = a -> Q

  assert Q [T= P
  ```

  You will find that FDR finds the errors on lines 1 and 2 of `P`, but not the one on line 2. This is because the processes `c -> STOP` from which `c` appears are the same on the first two lines, and the first has the shorter trace; while the error on line 3 happens from state `c -> c -> STOP` which is different. Thus FDR reports the error trace `<a,c>` twice but `<a,a,c>` not at all. See Exercise **??** for some ways of discovering multiple traces to the same trace.
- Compact controls the way in which FDR stores large state spaces. The higher the level of compression, the less space it should use. There is no need to worry about this setting unless you are running very large checks.
  In general there is a trade-off between CPU time and memory bandwidth when you run large checks on FDR. You can roughly imagine FDR as a process that has to shovel large amounts of previously stored data through the CPU, while it generates more data in large quantities that have to be stored for later. Its algorithms are tuned to make maximal use of the data it brings in, but the sheer quantity of this information sometimes means that the memory systems (and backing store when necessary) cannot deliver data fast enough to the CPU to keep it busy. By storing the data in a compressed

form, it can be got to the CPU faster, and the extra work the CPU has to do in expanding and re-compressing the data is often well worthwhile.
The default on this setting is at present normal (meaning that instead of storing each successive state separately, only those bytes of the next state that are different from the present one are stored, along with a table showing which bytes are different.) With present computer architectures, this almost always seems to give better speed (as well as memory consumption) than none. The high setting uses `gzip` on blocks of states, using typically around 70% of the memory required by normal. This tends to win when running a check that extends into backing store (virtual memory) because it both defers the point where that happens and means that less data needs to be traded with it when it is needed.

The Interrupt button stops whatever FDR is doing and attempts to retain information from previous completed checks if you ask it to Continue. Help brings up an online version of the FDR manual, which contains a far more complete description of what the tool does than we have given in this section. Your attention is drawn to the appendices on *Direct Control of FDR* and *Configuration*, which respectively describe FDR's *batch mode*, in which it is directly driven from command line rather than GUI, and ways of configuring your system (e.g. through environment variables) to guide FDR in its work. Batch mode is invaluable to anyone wishing to write an alternative front end to FDR, for example to support a different input language than CSP.

Defining non-process objects

The second section of `fdr_intro.csp` is a brief illustration of some of the styles of programming that can be used on non-process values. In a normal script functions like these would be used to help build process descriptions.

As we said earlier, the basic style of programming in CSP$_M$ is *functional* (or *declarative*): this means that a script consists of a series of definitions, frequently of functions, in which the value of a term can be evaluated by systematically *reducing* an expression. One of the great benefits of this style is that it allows one simply to write down the same sort of definitions of events, functions and sets in a CSP$_M$ program that one would in writing CSP down with paper and pencil. It also brings with it a type discipline that is invaluable in creating real system models, and which earlier versions of CSP notably lacked.

No special skill in creating functional programs is required for this style of use, but it makes CSP$_M$ a hugely expressive language for those who do acquire expertise in it, as is illustrated, for example, by the way we write a compiler in CSP in Chapter 2. Readers interested in this aspect of CSP$_M$ are advised to study a text on the programming language Haskell [?] or a related language, such as that by Richard Bird [?].

The following are the examples from `fdr_intro.csp` that illustrate some basic functional programming techniques, some of which are not available in Haskell.

```
factorial(0) = 1
factorial(n) = n*factorial(n-1)

reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

reverse'(<>) = <>
reverse'(t^<x>) = <x>^reverse'(<x>^t)

qsort(<>) = <>
qsort(<x>^xs) = let
                 p(<>,ls,es,gs) = (ls,es,gs)
                 p(<y>^ys,ls,es,gs) = if y < x then p(ys,<y>^ls,es,gs)
                             else if y == x then p(ys,ls,<y>^es,gs)
                                        else p(ys,ls,es,<y>^gs)
                 (lsx,esx,gsx) = p(xs,<>,<>,<>)
                within
                qsort(lsx)^<x>^esx^qsort(gsx)

qsort'(<>) = <>
qsort'(<x>^xs) = qsort'(<y | y<-xs, y<x >)^<x>^<y | y<-xs, y==x >^
                 qsort'(<y | y<-xs, y>x >)


facs = {(x,factorial(x)) | x <- {0..9}}

fact(n) = pick({m | (n',m) <- facs, n'==n})

invfact(m) = {n | (n,m') <- facs, m'=m}

pick({x}) = x
```

The definition of the factorial function (the number of ways one can arrange n different objects into a list) given here is essentially what you will find in a mathematics textbook. The thing to notice here is that there are two lines of definition, which act together using a technique called *pattern matching*. If you call this function with argument `0` it will match the pattern in the first line (you can use any fixed integer as a pattern). If you call it with any other argument, that match will fail and it will go onto the second line, which in this case always succeeds. So, for example, the term `factorial(3)` reduces successively to `3*factorial(2)`, `3*(2*factorial(1))`, `3*(2*(1*factorial(1)))`, `3*(2*(1*1))`, `3*(2*1)`, `3*2` and `6`.

As in most functional programming languages, lists are an important part of CSP$_M$, where the notation used for them is based on that of traces in CSP. Thus lists are written `<>` (the empty list), `<a,b,c>` or using *list comprehensions* such as `<n*2 | n <- <1..10>, n != 4>` (meaning twice each integer in the range `1..10` except for `4`). In a list comprehension, the right hand side consists of zero or more *generators* in which members are drawn from lists, and zero or more *predicates*, namely boolean expressions that allow us to select which of the generator values creates a member of the list.

The order in which things appear in lists is important, as is the number of times they occur. So `<a,b>`, `<b,a>` and `<b,a,a>` are all different. In a generated list, the second generator goes round once for each value of the first, the third goes round once for each pair of values of the first two, and so on. Thus `<10*i+j | i <- <0..9>, j <- <0..9>>` is the same as the list `<0..99>`.

Just as in many functional programming languages, CSP$_M$ permits the use of *infinite* lists such as `rep(x) = <x>^rep(x)`, though infinite lists cannot, for example, be used as parts of events.

The next two definitions, `reverse` and `reverse'`, are two different ways of defining the same function by pattern matching on lists. In CSP$_M$ you can pattern-match on the last (or several last) members of a list, as well as the first.

Probably the most practically important work Tony Hoare has done was not to invent CSP but to invent the *Quicksort* sorting algorithm.[2] The next two definitions (`qsort` and `qsort'`) both implement basic versions of this: to sort a nontrivial list they use the first member of the list as the *pivot*, partition the list into those things less than, equal to, and greater than, the pivot, and the result is then got by sorting those less than the pivot, then writing down the pivot and any other things equal to it, and finally sorting those things greater than it. The only difference between `qsort` and `qsort'` comes from the way the partitioning is done. `qsort` does this by creating a locally-defined (i.e. using `let .... within`) partitioning function `p` that goes through the members of `xs` one at a time and adds them to the three lists `ls`, `es` and `gs` as appropriate.

The amazingly concise definition of `qsort'` uses list comprehension to define

[2]See Note 1 at the end of this chapter for a salutary tale of how Quicksort's use in FDR gave an unexpected result.

these three lists separately. This is less efficient in terms of the number of reductions because of the way it requires three separate passes along `xs`: one for each of the analogues of `ls`, `eers` and `gs` rather than a single pass using the function `p` (for partition).

The final two definitions illustrate that, unusually, CSP$_M$ allows you to program with sets. The language allows you to have infinite sets like `Int` (all integers), but in practice there is not much you can do with these finitely. So, in contrast to lists, it is usually wise only to define finite sets in your scripts.[3] Sets have been included because they appear so much in CSP programs (for example as process alphabets and sets to be hidden). In a set, the order in which elements appear and the number of times each is put in are ignored: the sets `{1,2}`, `{2,1}` and `{1,1,2}` are all the same. This means that in order to do much with a set, the language implementation has to remove duplicates and to store the members in an order that depends only on their fully reduced forms. It is therefore not recommended to put potentially "infinitary" objects such as processes, functions and infinite lists into sets. These are just the same types of object that cannot be communicated over channels.

The first of the set-based definitions illustrates the standard way of turning a function (here `factorial`) into its *graph*: a set of pairs of function arguments and results. By taking a finite part of a function's graph like this one turns it into something that *can* sensibly be put into a set or communicated across a channel.

We then see how set comprehensions, which follow the pattern of list comprehensions (though all the generators now range over sets rather than lists), can be used to achieve the effect of applying and inverting the graph of a function. Notice that the inverse used here takes a member `m` of the domain of a function and maps it to the *set* of all values that map to `m`. In this case there is one value, 1, that has more than one inverse and many that have none.

The function `pick` uses pattern matching to take the unique element from a singleton set.

There is an inbuilt function `set(xs)` that converts a finite list `xs` into the set of its elements. There is no standard function that converts the other way, or indeed one that extracts an "arbitrary" element of a set `S`, because these processes are not *functional*: there are many ways to achieve them, so the language behaviour would be under-specified. See Exercise **??**, however.

The final part of `fdr_intro.csp` illustrates the use of user-defined data types,

[3]In common with Haskell, the implementation of the CSP$_M$ language is based on *lazy evaluation*, which is what allows to compute with infinite lists. The need to determine which the distinct elements of a set are means that much more evaluation is forced early when using sets as opposed to lists.

which are usually used at the level of program "data" and indexing sets. The effect of a declaration like

```
datatype fruit = Banana | Apple | Orange | Lemon | Mango
datatype cheese = Cheddar | Brie | Cheshire | Stilton | Camembert
```

is to introduce some new constants into the program. Values like `Orange` and `Stilton` are now things that can, for example, be passed along channels that have types `fruit` or `cheese`. (An often-followed convention is to give the labels introduced by data type definitions a capital letter, as here, but it is not compulsory.)

We can use types when constructing other types, as in

```
datatype item = Cheese.(cheese,Int) | Fruit.(fruit,Int)
              | Butter.Int
              | Mustard | Crackers
```

So we can identify a shopping basket with `List(item)`, the type of lists of `item`, each of which is either an item bought by weight (the `Int`) or one bought as a unit (such as a jar of mustard). So a typical shopping basket is

```
<Mustard,Crackers,Crackers,Butter.1,Cheese.(Camembert.2),Cheese.(Brie.2)>
```

`datatype` definitions can be recursive. For example the type

```
datatype fruitlist = EmptyFL | ConsFL.fruit.fruitlist
```

builds a type very similar to `List(fruit)`: in fact the two types are naturally equivalent, though different notation is required to use them. The labels introduced in `datatype` definitions need to be different from each other (so one cannot use one in different types) and from the other identifiers such as channel names and processes used in the script. For example we could not have used labels `Empty` and `Cons` in the above and

```
datatype cheeselist = EmptyCL | ConsCL.cheese.cheeselist
```

One can also construct structures that are more complex than simple lists, for example binary trees of integers and arbitrary trees of Booleans

```
datatype bintreeI = LeafI.Int | NodeI.bintree.bintree

datatype treeB = LeafB | NodeB.Bool.List(treeB)
```

Notice that `bintreeI` has integers at its leaf nodes, whereas `treeB` has Booleans at its non-leaf nodes. We will make major use of tree-like types in Chapters **??** and 2.

Functions of such datatypes are frequently defined using pattern matching:

```
fringe(LeafI.n) = <n>
fringe(NodeI.t1,t2) = (fringe(t1))^(fringe(t2))
```

is a function from `bintreeI` to `List(Int)`.

It is possible to use the type `Proc`, meaning processes, in defining a `datatype`, and we will see an example in Section **??** below. Members of such types cannot be communicated along channels and should not be used in sets.[4]

The user can also give names to types constructed straightforwardly, without the need for tags such as `Banana` or `LeafI`. For example

```
nametype MyInts = {MinI..MaxI}
nametype Basket = List(item)
nametype CountedItem = (item,MyInts)
```

The last of these is the type of pairs whose first element comes from `item` and whose second comes from `MyInts`.

The limits of FDR

The third and final section of `fdr_intro.csp` illustrates the boundary between what FDR can and cannot do.

The first thing FDR does is to identify a number of (usually sequential) component processes that it has to *compile* to *explicit state machines*, namely lists of states, where each state is marked with the initial actions (visible and τ) it can perform, together with the indexes of the states it can reach under each action. Bear in mind that sometimes there are many states a process can reach under τ (e.g. `P |~| Q`) or visible (e.g. `(a -> P)[](a -> Q)` actions. In the case of a refinement check, it has to do this both for the specification and the implementation.

It cannot complete its task if any of these components are either infinite state or have too many states to compile in a reasonable amount of time and space. This is illustrated by the following processes:

```
Inf1(n) = a -> b -> Inf1(n+1)
```

[4]User defined data types that include the process type `Proc` have only been supported since FDR 2.90.

```
Inf2 = a -> Inf2'(0,0)
Inf2'(n,0) = a -> Inf2'(n+1,n+1)
Inf2'(n,m) = b -> Inf2'(n,m-1)

Fin(0) = a -> b -> Fin(0)
Fin(n) = a -> b -> Fin(n-1)
```

`Inf1(0)` is compiled as an infinite state process – because its parameter value increases unboundedly – even though it is actually equivalent as a CSP process to the finite-state process `Fin(0)`. The parameter is causing the problem, even though it is irrelevant to behaviour: similar issues would arise if we allowed a parameter to vary infinitely even though only a finite number of values were relevant such as the `Bound(0,m)` below, in which the first parameter varies infinitely even though only the range `{0..m}` is relevant. The second version is equivalent but only finite state.

```
Bound(n,m) = a -> Bound(n,m+1)
             [] (m<=n)& b -> Bound(n,m)
             [] c -> Bound(0,m)

Bound'(n,m) = a -> Bound'(n,if n==m then m else m+1)
             [] (m<=n)& b -> Bound'(n,m)
             [] c -> Bound'(0,m)
```

A different manifestation of this same problem is that FDR has difficulty in handling processes that use large data types, particularly when many unrelated values of such a type can be held at the same time. Thus a process that inputs a member of a very large (or infinite) type may well be too large to compile. So while it can handle processes that hold a quite a lot of binary or similar values, or one or two values from a moderate-sized type and (thanks to the sub-process language) do complicated things to these values, FDR is not a tool for analysing the correctness, for example, of operations on the whole integer type.

There are some fairly advanced techniques that can allow us to extend results about small types to large ones: see Chapter **??**, for example. However all component processes that FDR actually compiles must use only small to moderate types (or similar sized parts of large or infinite ones), and furthermore must have all their parameters fully defined. One cannot, using a simple FDR check, verify a property of some process `P(n)` for *arbitrary* `n`, just for *particular* `n` such as 4 or 10.

Some of the types commonly used in CSP$_M$ programming can get large or infinite, besides `Int`. These include *lists* (there being n^k lists of length k over a type of size n) and *sets* (there being 2^n subsets of a set with size n). So, for example, while the process

```
channel isin, add, remove: T

SET(X) = add?x -> SET(union(X,{x})
         [] isin?x!member(x,X) -> SET(X)
         [] (card(X)==0) & isempty -> SET(X)
         [] remove?x -> SET(diff(X,{x}))
```

can be used provided the type `T` is reasonably small, by the time it has size approaching 20 the number of states will start to get slow to compile. FDR is much slower at compiling each state of a component process than it is at running the final check, so while most checks of size 1,000,000 will take just a few seconds on a modern computer, compiling a component of that size will take much longer. We will see how this fact can be used to produce a better version of `SET` in Section **??**.

Exercise 1.1.1

If `X` is a set drawn from a type `T` that can be compared (i.e. either the inbuilt relation `<` is meaningful or you can define a similar linear comparison function yourself, then it is possible to turn `X` into a list by *sorting* into ascending order using the CSP$_M$ language. Find a way of doing this. [Hint: you can write a version of Quicksort based on sets provided you can find a way of choosing the pivot, which can be done by finding a subset of `X` with size 1 and applying `pick` to it. Note that you can count the size of a finite set with the function `card(.)`.]

Exercise 1.1.2 One of the author's favourite ways of using FDR is in solving combinatorial puzzles. Most of these require parallel processes for a reasonable model, but the following one can be solved quite naturally and quickly with a sequential one.

> A farmer has a wolf, a goat, and a large cabbage on one side of a river, and a small boat. The boat is only big enough to hold the farmer and one of this group. He fears that, if he were absent, the wolf would attack the goat and/or the goat would eat the cabbage if they had a chance. He therefore cannot leave a group containing either of these pairs on either side of river. *How does he get all three across, intact?*

Write a CSP$_M$ description of this puzzle. It should keep parameters recording what is on each side of the river, and where the farmer is, and should have events `fwd`, `back`, `fwdwith.x` and `backwith.x` for `x:{Wolf,Goat,Cabbage}`, where `fwd` and `fwdwith.x` represent the farmer rowing across the river alone of with one of the group; and similarly for `back` and `back.x`. If the puzzle is solved your process should communicate the event `done` so that we can test via trace refinement whether the puzzle is solvable using the refinement check `CHAOS(diff(Events,{done}) [T= Puzzle`.

1.2 Checking parallel processes

While we can see the basic functionality of FDR in checking sequential processes, it is not until we start analysing parallel systems that it shows its power. That is both because understanding how parallel systems behave is inherently complex and because the number of states that a system has can grow exponentially with the number of parallel components.

The states of a sequential program like the ones we discussed in the previous section are relatively easy to understand. At any one time it is always *at* some particular state that comes from its definition. Thus the states of the process `SET({})` described above are just `SET(X)` as `X` varies over the subsets of `T`, and the states of `Inf1(0)` are all of the form `Inf1(n)` or `b -> Inf1(n)` for `n` some integer.

In a parallel process, on the other hand, there will several or many separate processes, each *at* some separate control state, so a state of the process `|| i:{1..m} @ [A(i)] P(i)` is described by an `m`-tuple of states, the `i`th being a state of `P(i)`. Understanding the behaviour of a compound system like this may be far less clear, and will probably have many more states, than that of a sequential system.

We can implement an equivalent of the `SET(X)` process in a way much better suited to FDR, using parallel. Define:

```
Present(x) = isin!x -> Present(x)
             [] add!x -> Present(x)
             [] remove!x -> Absent(x)

Absent(x) = isempty -> Absent(x)
            [] add!x -> Present(x)
            [] remove!x -> Absent(x)

ParSet(X) = [|{isempty}|] i:T @ (if member(i,X) then Present(x)
                                                 else Absent(x))
```

Notice how this parallel combination implements a process equivalent to `SET(X)` (something you can verify for smallish `T`). It has exactly as many states as `SET(X)`. You can count the states of any CSP process `P` at all by running the check `CHAOS(Events) [T= P` This is certain to succeed, and the number of states reported in the `Status` window is always the number of distinct states FDR finds in `P`. In some cases this may give a smaller number you expect because of the way FDR automatically applies a simple compression operator to each component process, but both versions of our set process give 2^k for k the size of `T`.

So why do we say that the parallel implementation is better for FDR? The first reason, as will be apparent from running the counting checks for appropriate sizes of `T`, is speed. As remarked earlier, FDR is much faster at running checks than compiling a similar number of states of component processes. The `ParSet` version gives only `2*card(T)` states to compile, so almost all the time spent counting states for it is spent in the running (or model checking) phase of the check, whereas the sequential version `SET` spends almost all its time compiling.

Unfortunately, so dire are the effects of the exponential explosion in state space that this example brings, that `ParSet` does not increase the tolerable size of `T` by very much: 30 would give it just over a billion states, which takes far too much time and CPU to waste on such a pointless check!

It is, however, frequently possible to use `ParSet` and similar processes in contexts where there are very many more components than that. What matters in an actual run is how many different subsets are encountered during a run, and making a set process part of the state that other processes can use as a service may well substantially limit the number of subsets that are actually explored.

As a simple example, consider the following program, which allows an ant to move around a rectangular board provided it never steps twice on the same square except the one where it started. The latter property is enforced with a process based on `ParSet` that now only allows a `done` event in place of `isempty`, and an action that removes a member from the set, which is naturally identified and synchronised with the action by which the ant moves to the given place in the grid.

```
T = {(m,n) | m <- {1..M}, n <- {1..N}}

channel move:  T

channel done


Ant((m,n)) = (n>1)& move.(m,n-1) -> Ant((m,n-1))
             [] (n<N)& move.(m,n+1) -> Ant((m,n+1))
             [] (m>1)& move.(m-1,n) -> Ant((m-1,n))
             [] (m<M)& move.(m+1,n) -> Ant((m+1,n))

Unused(x) = move!x -> Used(x)

Used(x) = done -> Used(x)

Board(X) = [|{done}|] i:T @ (if member(i,X) then Unused(i)
                                        else Used(i))

System = Ant(start) [|{|move|}|]Board(T)
```

Notice that the ant crawls only to directly adjacent squares. The `System` process has many fewer states than `Board(T)` for the simple reason that most subsets of the grid are not the sets of points of some non-revisiting path starting from `start`

So, for example, the 7×6 version of this system has 261M (approximately 2^{21}) states which, though a lot, is hugely less than the 2^{42} states of `Board(T)` here. What we find, and this is typical of examples from many domains, is that only a small fraction of the potential states of this subprocess are required in the real system.

The way FDR works means that, when we use the `Board` representation, all the extra states are never visited. However if a single sequential component like `SET` is used, all the states are compiled, whether used or not.

For both these reasons it is almost always a good idea, if you have the choice, to factor what would otherwise be a large component process into a number of components running in parallel. In doing this you should, where-ever possible, avoid introducing extra states. We will see an interesting but more advanced technique for dealing with this last issue in Section **??**.

A *Hamiltonian circuit* is a path that visits every location in some graph exactly once and then ends up back where it started. The specification `NoHamiltonian` below says that there is no Hamiltonian tour around our grid: the event `move.start` cannot immediately be followed by the `done` event that indicates that there are no squares left to visit. It is clear that whether this specification is true or not does not depend on which `start` location is chosen, but it does depend on the dimensions `M` and `N`.

```
NoHamiltonian = move?(i,j) -> if (i,j)==start then NoHamiltonian'
                              else NoHamiltonian
                [] done -> NoHamiltonian

NoHamiltonian' = move?_ -> NoHamiltonian

assert NoHamiltonian [T= System
```

EXERCISE 1.2.1 Recall the example in Section 1.1.2 that showed how FDR only reports a single trace to each state that exhibits an erroneous behaviour. You can get more traces that lead to a single trace by putting your original system in parallel with a second companion process that (i) does not change the behaviour of the system and (ii) changes its own state sufficiently on the trace to get FDR to observe different states of the compound system despite the main process being in the same states. The only real difficulty in doing this, provided the main implementation never terminates, is to ensure that the states of the companion process are not identified by strong bisimulation. Perhaps the best way of doing this is to construct the companion so that it always accepts every event of the process it accompanies, but sometimes is able to communicate extra events too. The extra events are then prevented by the way the system is put in parallel, say `Impl[|Events|]Companion)[|Extra|]STOP`. Note that this technique will, by its very nature, increase the state-space of a check, and that the companion process needs to be finite state.

(i) Find a companion process with alphabet `{a,b,c}` that can find the trace `<a,a,c>` in the quoted example.

(ii) Find one that can find multiple Hamiltonian paths (when they exist) with the example above noting that all such paths are necessarily the same length.

The structure of a refinement check

Having got this far through this chapter, you should already have run some largish checks on FDR. If you open up the `Status` window you will see the stages of these checks being reported on. There are three phases to each refinement check.

Compilation (in which with the `auto` level of messages what you see reported is the number of transitions calculated so far) is the stage at which, in the check of a parallel process, FDR identifies the parallel components and compiles these to explicit state machines as discussed on page **??**. It also works out a set of rules, called *supercombinators* (see page **??**) by which the transitions of the parallel combination can be deduced from those of the components. It does this both for the specification and implementation processes.

Normalisation is the phase during which the specification process is transformed into a form that can efficiently be compared against the implementation. In essence this is a form such that, for every trace, there is a unique state that the specification is in. In the case of simple specifications this is so quick and easy that there is no evidence for this phase other than FDR reporting that it is starting and ending.

The user has to be aware, however, that if a complex specification is normalised this can take a long time, and in some cases can create normalised processes that are very much larger than the original specifications. In particular, before a process is normalised it is turned into an explicit transition system and so even if normalisation goes very smoothly one is still only able to handle specification processes that are very much smaller than implementations. (At the time of writing, the author would think twice before using a specification process with more than ??? states.) Full details of the normalisation algorithm can be found in ???: in the untypical worst case, as demonstrated by the example file `pathol.csp`, it can generate a normal form representation that is exponentially larger than the original process (in terms of state space).

The closer a specification is to being in normal form, namely having only a single execution path leading to each state, the easier it is to normalise. Complexity can rise dramatically when decisions made early in a trace can give rise to separate ways of performing it. Fortunately, almost all processes written as clear specifications have essentially only one way of performing each trace, and so cause no problems at all.

In practice, you are only likely to have problems normalising the left-hand side of a refinement check when it was either constructed as an implementation in its own right, or is some modified form of such a process. The first of these occurs when you are using refinement in its more obvious sense of testing whether one prospective implementation can be used in place of the other. The second of these appears in some advanced form of specification, some of which can be found in Chapter **??**.

By the time normalisation is complete, we have efficient representations both of the implementation and a form of the specification that is efficient to check against.

The main "running" or "model-checking" phase of the check can now take place. What it does is to explore the pairs of states that can be reached on any trace by the normalised specification and the implementation. In each such pair the specification state imposes restrictions on what the implementation state can do: these will relate to the next events, and for richer models to things like refusal sets and divergence information. Usually each implementation state ends up paired with only one specification state, but there may be more. Consider the following:

```
AS = a -> AS

Spec = a -> a -> a -> Spec
       [] b -> b -> STOP
```

Clearly `AS` has only one state whereas `Spec` (presented here in what is already normal form) has 5. The check `Spec [T= AS` naturally succeeds, and explores 3 state pairs. After traces `<a,a,...,a>` with length divisible by 3, the implementation is allowed to perform either `a` or `b`, but on all other lengths it is not. Of course it never does, so the single implementation state is always paired with one of the top three states of `Spec`.

When a refinement check completes successfully, it actually represents a proof that, for each state pair `(SS,IS)` explored (respectively the specification and implementation states), `SS [X= IS` where `X` is the relevant model.

In its standard mode of exploring a check, the running phase is performed as a breadth-first search in which each state pair is considered separately. In other words FDR first finds the successor states of the starting state (the pair of the initial states of specification and implementation), then their successor states, and so on. It explores all of the successors at level n (i.e., the pairs reachable in n actions but not in any fewer) before going on to explore those at level $n+1$. This approach has the advantage that, when a check fails, FDR finds one involving the fewest possible actions: the error reported by the debugger will have the fewest possible steps in it. These steps include τ actions, so it does not necessarily report the shortest possible *trace*.

The print-out you see in the Status window at this stage is just a count of the number of state-pairs completed, which in the standard checking mode are broken down into the different levels of the search. These levels always start off small, and then typically grow until they reach a peak and then shrink back until eventually no more new state pairs are found and the check terminates. The rate at which this growth and shrinkage happens depends, of course, on the details of the CSP processes you are checking.

Usually, if a check fails, a counter-example will be found somewhere in the middle of the enumeration of the state pairs, more often near the beginning than the end. The main exceptions to this rule are checks designed to search for a particular example, such as the solution to a puzzle. In these cases the counter-examples are frequently found at or near the end. This is true of the checks designed to search for Hamiltonian circuits since necessarily such a circuit cannot be found until all the points in the grid have been used up.

If a counter-example is found, the user is given the chance to examine the behaviour that caused it. In its standard debugging mode, a window will appear in which the trace leading to the error is displayed together with refusal or divergence information if relevant. For refusals, you have the option to view *acceptance* information, since this is frequently more concise and intuitive. (A minimal acceptance is the complement of a maximal refusal.)

FDR also gives you the opportunity to explore the contributing behaviours of the various subcomponents of the implementation. So for example, going down through a hiding operator will show you the original identities of the hidden events that are τs at the top level, and you will be able to see what each component of a parallel composition did.

All of this information is about the implementation process. It is easy to attribute blame on its side of the refinement check, since it has performed a single behaviour that is to blame for violating the specification. Sometimes, however, when you look at the debugging information, you will come to the conclusion that the implementation behaved perfectly: you want to understand why the specification did not allow it. Working out why one process *did not* do something is a great deal harder, and a lot less systematic, than working out how the other one *did* it. Unfortunately, therefore, FDR offers no direct support for the debugging of specifications.

There are many further advanced functions of FDR that we have not described here. Some of these are described in Sections (inc last of this chapter)???. Hopefully yet more will be developed after the publication of this book.

Exercise 1.2.2 Build a parallel version of the Wolf/Goat/Cabbage puzzle (Exercise **??**). There should be separate processes for each of the Farmer and his three "companions", which should be aware of which side of the river they are on (i.e. each should have two basic states), and which are able to synchronise on the events described in the previous exercise that they are involved in. (So each involves the Farmer, and the cabbage only participates in `fwdwith.Cabbage` and `backwith.Cabbage`.) There should be additional events `wolf_attack.s` and `cabbage_eaten.s` for `s` being the side where the event happens, each of which is synchronised by the two parties obviously involved and the Farmer when he is on the opposite side of the river.

Obviously one would expect either sort of attack to be possible in this model, and you should formulate FDR checks to prove this.

By adding extra processes into the parallel composition, impose the obvious safety constraint on the Farmer, and find the solution to the puzzle.

Exercise 1.2.3 This exercise is about the Hamiltonian circuit check.

1. Why is it obvious that the value of `start` does not affect whether the check for such a circuit succeeds or not?
2. Use FDR and different values of the grid dimensions M and N to formulate a conjecture about what values of these parameters allow a Hamiltonian circuit. Then use mathematical analysis to prove it. [Hint: imagine that the grid has its nodes coloured black and white like a chess-board. If the starting square is black, what is the colour of the last member of the circuit before it returns to the start?]

1.3 Failures and divergences

All the refinement checks we have discussed to date involve the traces model: one process refines another if its set of traces is contained in that of the other. For various reasons a large percentages of practical FDR checks are trace checks, but richer models allow the tool to check more detailed properties.

The standard checks for deadlock and livelock freedom are in fact refinement checks performed in the richer models, and the determinism check implemented by

FDR conceals such a check. In many cases users are content with establishing the first two of these properties, without delving more deeply into what failures and failures/divergences properties hold.

The way the failures-divergences model $\mathcal{N}$ treats divergence strictly suggests that any check performed in this model should regard divergence as an error to be eliminated. That is very often the case, and then the inbuilt divergence-freedom check is all one has to do in that model.

By and large it is better to do this and then base subsequent checks in models that do not model divergence, for a purely pragmatic reason. That is because, every time FDR performs a refinement check in the failures-divergences model, it has to test each reached state for divergence. This is an additional, and sometimes expensive, additional piece of work, and it is better to do this once rather than each time a property of a given system is verified.

Divergence can sometimes be useful in formulating specifications. If we want to make a forcible *don't care* statement in a specification, namely the implementation is allowed to anything at all once it performs a particular sort of trace, a good way of doing this is by putting a divergent term in the specification on the left-hand side of the refinement check. So, for example, if we have a specification `Spec` and we want to transform it so that it allows the implementation to do anything at all after the event `c`, we can do this by using[5] `Spec[|{c}|](c -> DIV)` over $\mathcal{N}$. This particular example makes interesting use of the strict treatment of divergence in $\mathcal{N}$.

The use of `DIV` as "don't care" brings a real practical benefit in cases where we expect much of the implementation's behaviour to come into the category where this clause will apply. FDR recognises, when checking over $\mathcal{N}$, that a divergent specification state is refined by anything, and it does not pursue the search in the running phase of a check when a pair with a divergent specification state is reached. In such cases refinement can be proved without visiting all of the specification states. At the time of writing there is no similar optimisation for refinements not involving divergence, in part because the least refined process in such models is harder to recognise.

In understanding processes it is sometimes useful to ask whether a process with more than the "natural" set of events hidden can diverge. For example, if `P` is divergence-free but `P \ A` is not, we know that `P` can perform an infinite sequence of actions from the set `A`. There is an example of this idea in Section **??**.

We will now concentrate on failures checking `[F=`. The check `P [F= Q` says two different things depending on whether or not `Q` is known to be divergence free. If it is, then it implies `P [FD= Q` which says that whatever trace `Q` has performed,

[5]This version only allows those `c` events that are allowed by `Spec`. See Exercise **??** for an alternative version.

it will eventually reach a stable state in which it makes an offer acceptable to P. In other words, P will reach a stable state on the same trace where the set of events it offers is contained in the set now offered by Q. So P will definitely accept a member of any set of events that Q cannot refuse. So, for example, if P is a deterministic process and Q is divergence-free, P [F= Q says that P and Q behave identically.

Such a check allows us to draw positive inferences about the *liveness* of Q. If, on the other hand, we suspect Q might diverge, then we can make no such inference; after all, P [F= DIV for every process P. If we know Q is divergence-free, then proving P [F= Q\X proves the traces of Q\X are contained in those of P, and that every stable state of Q *that is not offering any member of* X makes an offer acceptable to the corresponding state of P. So, for example, DIV [F= P\X says P never performs any event outside the set X *and* never deadlocks, and if AB = a -> b -> AB and C = diff(Events,{a,b}) then

```
AB [F= (P [|C|] CHAOS(C))\C

AB [F= P\C
```

say two different things on the assumption that P is divergence free. The first says a and b alternate (possibly interrupted by other events), and that P is *always* offering one of these two events. The second simply says they alternate and P is deadlock-free.

Variations on this theme can express many different properties.

Sometimes we might model a system where we know divergence is formally possible in a system P, at least with the CSP description we have given it, but we want to analyse the system on the assumption that this divergence never happens. A good example of this is the model of the Alternating Bit Protocol in Section **??** without any bound on the numbers of errors that can occur: making this assumption there corresponds roughly believing that not enough errors happen to cause divergence. We will revisit this example in Chapter **??**, which will also give many more of the same nature. In cases like that, we can analyse our process P by proving it refines failures specifications and in effect be proving the same properties over $\mathcal{N}$ under the *fairness* assumption that excludes divergence.

Exercise 1.3.1 We showed above how to weaken a specification Spec so it makes no assertion about the implementation at all when the implementation has performed a c that Spec allows. Show how to weaken it further so that it now always allows a c in addition to this. Your revised version should always allow the *event* c, but the revised specification should have identical refusal sets to the only one after every trace s that does not include a c.

What refinement relations hold between the three versions of the specification (the original, the modification on page **??**, and your new one)?

EXERCISE 1.3.2 Recall how, on page **??**, we gave an example of a one-state implementation process `AS` that gets paired with three of the five states of a normal form during a successful trace refinement check. Use the additional expressive power to failures to find an N-state normal form `Spec'(N)` (for arbitrary `N>0`) which this same process `[F=` refines and where it gets paired with every single normal form state in the refinement check.

If `M>1` is coprime to `N` (i.e. their least common multiple is `M*N`), find an `M`-state process `AS'(n)` that cycles through its states using the action `a` and which is a *proper* stable failures refinement of `AS` but has the same traces.

If you check `Spec'(N) [F= AS'(M)` you may find that `N*M` state pairs are reached. (If not, find an alternative `AS'(M)` that does have this property.) This is, of course, the greatest possible number given the sizes of the two processes. Explain why this happens and why it is helpful to assume that `N` and `M` are coprime.

EXERCISE 1.3.3 Formulate refinement checks that establish the following properties of divergence-free processes `P`:

(i) Every infinite trace of `P` contains a `b`.

(ii) Every infinite trace of `P` contains an infinite number of `b`s.

(iii) Every infinite trace of `P` contains the finite trace `s` as a (not necessarily consecutive) subsequence. [This part and the next are more difficult than they look. Check that that `AS = a -> AS` does not satisfy the specification you derive for `s = <a,b>`.]

(iv) Every infinite trace of `P` contains the infinite trace `u` as a (not necessarily consecutive) subsequence.

1.4 Determinism checking

As described in Section **??**, the question of whether or not a process is *deterministic* is important in understanding it, and in particular can be said to decide whether it is *testable.*

One cannot check the determinism of a process `P` by refinement checking it against some specification in any model X: `Spec [X= P`, since the deterministic processes are the maximal ones under refinement, and the nondeterministic choice of all deterministic processes is very nondeterministic!

Nevertheless there are several ways of using refinement checking to decide if a process is deterministic. The FDR algorithm (i.e. the one invoked by asking FDR explicitly whether a process is deterministic) was invented by the author and proceeds as follows.

- Given a process `P`, extract a "pre-deterministic" refinement of it as follows, by extracting a subset of its states and transitions.:
  - Include the root node in the subset.
  - If an included node `Q` has a τ transition, then it has only a single transition, namely τ to node that it arbitrarily chosen amongst those reachable under τ from `Q`.
  - If `Q` is stable, then choose, for each event `a` it can perform, a single arbitrary one of the states `Q` can reach under `a`.
  - In both cases there is a practical advantage in picking an already-seen state if possible, but for the decision procedure this is not necessary.

  The result is a process that is deterministic on every state except those on on which it diverges.
- If divergence has been discovered then `P` is not deterministic.
- Otherwise, the subset represents a process `P'` that is deterministic and is a refinement of `P`. If the two are failures-divergences equivalent then `P` is deterministic; if they are not then it is nondeterministic (since a deterministic process has no proper refinements over $\mathcal{N}$).
- It follows that, when `P'` is divergence free, `P` is deterministic if and only if `P' [FD= P` or equivalently `P' [F= P`. Fortunately the way `P'` has been produced means it is trivial to normalise.
- When `P` has been found not to be deterministic then, except in the case where a divergence is found, the FDR debugger reports both the behaviour of `P` that led to refinement failing, and the unique path of `P'` to a stable state after the same trace.

This algorithm works efficiently and, when the refinement check is successful, will always visit exactly as many state pairs as there are states in `P`. This is because, by the time the refinement check is carried out `P'` has been reduced to a normal form all of whose states are deterministic and therefore incomparable: it follows that each state of deterministic `P` refines exactly one of them.

Because the FDR algorithm relies on a special routine to extract `P'`, you cannot reproduce this algorithm using its refinement checking function.

FDR actually gives you two options for determinism checking: you can pick the failures-divergence $\mathcal{N}$ or stable failures $\mathcal{F}$ model to decide it in. The version above is the former, corresponding to that described in Section -**??**. Formally speaking, determinism checking over $\mathcal{F}$ seeks to decide whether there is a deterministic process `P'` such that `P' [F= P`. Notice that `DIV` is *$\mathcal{F}$-deterministic* (as we shall call this property) since it refines *all* deterministic processes. In practice, however, this function is only used on processes that are known to be divergence-free, so that the two notions co-incide. There are two cases we need to consider.

The first is where running the FDR check for divergence freedom on `P` gives a positive answer. There is not much to say about this since it is exactly in the spirit we discussed in Section **??** of establishing first that a process is divergence-free and doing all further checks over $\mathcal{F}$ or $\mathcal{T}$ as appropriate.

The second is when checking for divergence freedom does find a counter-example, but where (as discussed in the same place) we are making some fairness assumption that means we can ignore such behaviour. In such cases we cannot be sure the above algorithm works, since we do not know what to do when the routine that extracts `P'` finds a divergence. What often happens in this case is that FDR, instead of giving its usual tick or cross sign to show a positive or negative outcome to a check, instead displays a "dangerous bend" sign to show that it failed to find an answer.

Since this second style of checking for $\mathcal{F}$-determinism has a major practical use in computer security (see Section **??**), this is not a happy position to be in. Fortunately there are two ways round it that can be used with FDR: one depends on the structure of the security checks and is set out in that section, and the second is to use Lazić's algorithm for determinism checking (introduced in [**?**]), which depends on running the target process in parallel with itself and a harness and refinement-checking the result against a simple specification. Lazić's algorithm, modified slightly to make it easier to use on FDR, is described below.

- For any process, let `Clunking(P) = P [|E|] Clunker`, where `E` includes all events that `P` uses, but not the special event `clunk`, and

  ```
  Clunker = [] x:E @ x -> clunk -> Clunker
  ```

  `Clunking(P)` therefore behaves exactly like `P`, except that it communicates `clunk` between each pair of other events.

- It follows that the process `(Clunking(P) [|{clunk}|] Clunking(P))\{clunk}` allows both copies of `P` to proceed independently, except that their individual traces never differ in length by more than one.

- If `P` is deterministic, then, whenever one copy of `P` performs an event, the other one cannot refuse it provided they have both performed the same trace to date. It follows that if we run `RHSDet(P) =`

  ```
  ((Clunking(P) [|{clunk}|] Clunking(P))\{clunk})[|E|] Double
  ```

  where `Double = [] x:E @ x -> x -> Double`, then the result will never deadlock after an trace with odd length. Such a deadlock can only occur if, after some trace of the form `<a,a,b,b,...,d,d>` in which each `P` has performed `<a,b,...,d>`, one copy of `P` accepts some event `e` and the other refuses it. This exactly corresponds to `P` *not* being $\mathcal{F}$-deterministic.

We can therefore check determinism and $\mathcal{F}$-determinism by testing whether `RHSDet(P)` refines the following process, respectively over $\mathcal{N}$ and $\mathcal{F}$.

```
DetSpec = STOP |~| ([] x:E @ x -> x -> DetSpec)
```

Because it runs `P` in parallel with itself, Lazić's algorithm is at worst quadratic in the state space of `P`. (In other words, the number of states can be as many as the square of the state space of `P`.) In most cases, however, it is much better than this, but not as efficient as the FDR check.

Lazić's algorithm works (in the respective models) to determine whether a process is deterministic (i.e. over $\mathcal{N}$) or $\mathcal{F}$-deterministic.

The fact that this algorithm is implemented by the user in terms of refinement checking means that it is easy to vary, and in fact many variations on this check have been used when one wants to compare the different ways in which a process `P` can behave on the same or similar traces. There is an example of this in Exercise **??** below, and we will meet more in Chapter **??**.

EXERCISE 1.4.1 Which of the following processes are $\mathcal{F}$-deterministic? You should try to decide which *without* FDR, and then try them out on FDR. (i) `a -> DIV`, (ii) `(a -> DIV)|~|(a -> STOP)`, (iii) `(a -> DIV) |~| (b -> DIV)`, (iv) `(a -> DIV) ||| ABS`, where `ABS = a -> ABS [] b -> ABS`.

EXERCISE 1.4.2 A further way of checking to see if a process is deterministic is to replace the arbitrary deterministic refinement `P'` by a specific one in the FDR algorithm. The most obvious one is the process with the same traces as `P` which never diverges and never refuses an event that `P` can perform after the current trace. This could be created by embedding the traces normal form of `P`, as the unique most deterministic process with that set of traces, within one of the richer models, though FDR does not offer a way of doing this at the time of writing. Compare this with the two algorithms offered above: does it work in general for $\mathcal{F}$-determinism? How does it compare in terms of worst case performance? How would you expect it to compare in the average case?

EXERCISE 1.4.3 Modify Lazić's check of determinism so that it checks the following property of a process: `P` cannot, after any trace s, allow the event `b` in `B` while also having the stable failure `(s,{b})`.

This is a weaker specification than determinism, since it allows `P` to show nondeterminism in events that are not members of `B`.

1.5 Compression

One of the ways in which FDR can help you overcome the state explosion problem is by attempting to reduce the number of states it has to visit by *compressing* some of the subcomponents of a system. Since the questions that FDR poses about processes are all posed in terms of semantic models like $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{N}$, it really does not matter how these processes are represented as transition systems. It follows that if, presented with the representation of some process as one transition system, it can find a smaller one with the same value in the relevant model, it is probably better to use the latter since it will be quicker to perform computations on.

FDR supplies a number of functions that try to compress a transition system in this way, most of them behaving slightly differently depending on which model we happen to be using. Three deserve a mention at this stage, and all will be described in more detail in Section **??**:

1. Strong bisimulation `sbisim` identifies two nodes if their patterns of actions are equivalent in a very strong way. FDR automatically applies this to all the low-level component processes it compiles, so there is no point in applying it again to these processes.

2. Normalisation `normal` is the process already discussed: the same function that is always applied to the specification. Its last stage is also strong bisimulation, so again there is no point in applying `sbisim` to its result. As described earlier, it may actually expand a process, so needs to be used carefully.

3. Diamond compression `diamond` can be viewed as a sort of half normalisation. It eliminates all τ actions and seeks to eliminate as many of the states of the original system as possible, in summary this is by recording, if the state `S` has many states reachable by chains of τs, as few of these "down-τ" states as possible and adding all of their behaviour to `S`. As a preliminary this function calls a function called `tau_loop_eliminate` that identifies all pairs of states that are reachable by τ-chains from each other. `diamond` does not call `sbisim` automatically, and one frequently gets good results from applying the combination `sbdia(P) = sbisim(diamond(P))`. `diamond` can only compress processes with τ actions, but it never expands the state space.

This last combination and `normal` are the two most used compression functions. On some examples each performs better than the other. As we will see in Section **??**, `sbdia` and `normal` always give exactly the same result when applied to a deterministic Proc es `P`.

There is no point (at least for speed) in applying a compression function to the complete right-hand side of a refinement check, or to a process being checked for deadlock, livelock or determinism (using the FDR algorithm). This is because it is much more difficult to apply any of these compressions to a process than it is to refinement check it. In fact, the first thing FDR does when applying any of the above functions is to turn the object process into an explicit state machine.

When working in models richer than $\mathcal{T}$, both `normal` and `diamond` can produce transition systems with more detail than an ordinary LTS: they are marked with *acceptance* and perhaps *divergence* information. This is because, without this, the compression would lose important details of how the object process behaves. In a *generalised* labelled system, every node that is not marked as divergent and which has no τ actions must be marked with one of more acceptance, namely a subset of its initial actions that is incomparable with all its other minimal acceptances. Other nodes *may* have acceptances.

We then understand that a process implemented by such a GLTS starts at the initial node, and can follow any route of actions through the system. At any node marked as divergent it can diverge (as though that node had a τ action to itself), or it can become stable, accepting actions only from A, if A is one of its acceptances. For most purposes it is legitimate and better to reduce any node's acceptances to *minimal acceptances* (namely ones that are not subsets of any other) because these display all refusals and therefore stable failures. Figure **??** shows the result of compressing the process on the left, the transition system of

```
(b -> b -> STOP)[> (b -> STOP) [> (a -> STOP)
```

under `normal` and `diamond` over $\mathcal{N}$. Notice that both, by eliminating τ actions, require acceptance marking to reveal that the only acceptance set in the initial state is `{a}`. `normal` gives a unique initial arc labelled by `b` whereas `diamond` gives two.

The author has found that using the `graph` mode of FDR to view the result of a compression is a good way of understanding how the various choices work, and how effective they are in particular examples.

To use non-CSP functions such as `normal` and `diamond` which are implemented directly by FDR and whose purpose is to transform one state-space representation of a process into another, equivalent one, they must be declared within the CSP$_M$ scripts that use them. The form of the declaration is

- transparent normal, sbisim, diamond

with the word `transparent` indicating that they are not intended to alter the meaning of the objects they are applied to.

1.5.1 Using compression

The purpose of using these compression functions is to reduce the number of states explored when checking or normalising a complete system. Since the cause of state explosion is almost invariably the multiplication that arises from parallel composition, the right place to apply compression is to a process that is to be put in parallel with another. These might either be individual sequential components or subnetworks consisting of a number of these in parallel. You can of course compress subnetworks built up of smaller ones that have themselves been compressed, and so on.

It is difficult to give any hard and fast rules about when compression techniques will work effectively. The author has frequently been surprised both at how well they work on some examples and at how badly they work on others. Some examples are given below, further ones can be found in Chapters ????, and the reader will find many more in the files to be found on the web site, but there is no real substitute for personal experience.

Broadly speaking, you have a reasonable chance of success when the following are all true:

(a) Part of the alphabet of the process you are compressing has been hidden, either because the hidden events are true internal actions or because they have been abstracted as irrelevant to the specification (following the principles set out in Section **??**).

(b) It usually helps if these hidden actions represent progress within the process, as opposed to the resolution of nondeterminism. See page **??** for a good example.

(c) It is not the case that a large part of the state space of the process you are compressing is never visited when it is put into the complete system. The danger here is that you will expend too much effort compressing irrelevant states, or that the irrelevant states will make the subsystems too large to compress. See Example 1.5.1 below for some ways to reduce or avoid this problem.

The following principles should generally be followed when you are structuring a network for compression:

1. Put together processes which communicate with each other together early. For example, in the dining philosophers, you should build up the system out of consecutive fork/philosopher pairs rather than putting the philosophers all together, the forks all together and then putting these two processes together at the highest level.

2. Hide all events at as low a level as is possible. The laws of CSP (in particular ⟨hide-$_X\|_Y$-dist⟩ (**??**)) allow the movement of hiding inside and outside a parallel operator as long as its synchronisations are not interfered with. In general, therefore, any event that is to be hidden should be hidden the first time that (in building up the process) it no longer has to be synchronised at a higher level. The reason for this is that the compression techniques all tend to work much more effectively on systems with many τ actions.

The typical system one wants to apply compression to often takes the form

```
System = (|| i:I @ [A(i)] P(i))\H
```

The above principles suggest that one should distribute the hiding of the set `H` through the composition using ⟨hide-$_X\|_Y$-dist⟩ (**??**), moving it all to as low a level as possible. The only thing that prevents the hiding of an event being distributed across a parallel operation is when it is synchronised. This is a non-trivial calculation, particularly since one will frequently want to hide further events in `System` in order to get efficient representations of some specifications, and might therefore have to do the job several times. For example one could decide whether `System` can ever deadlock or communicate `error` via the checks

```
SKIP [F= System\Events
SKIP [T= System\diff(Events,{error})
```

By far the easiest way of dealing with this is use utilities from an `include` file such as `compression09.csp` that does all these calculations for you. This supplies a number of functions that assume that the network is presented as a data structure involving *alphabetised processes*: in the above example these are the pairs `(P(i),A(i))`.

Depending on which function you want to use from `compression09.csp`, networks must be represented in one of two forms: a simple list of alphabetised processes `<(P(i),A(i) | i <- L>` with `L` being some ordering of the indexing set `I`, or a member of the data type

```
datatype SCTree = SCLeaf.(Proc,Set(Event)) | SCNode.Seq(SCNode)
```

In the latter a tree of processes is defined to be either a leaf containing a single alphabetised process or a sequence of trees (with the file assuming that all such lists are non-empty).

The intention with the type `SCTree` is to put a parallel system together in a hierarchical fashion, combining individual processes into groups, those into larger groups and so on until the network is composed. This is most natural – and we might hope to get the best performance from compression – when the processes grouped together interact a lot with each other and less so with ones outside the group.

One can, of course, turn a list of processes into one of these trees in various ways. For example, given a branching factor `K` we can define

```
balanced(K,<p>) = SCLeaf.p
balanced(K,ps) = SCNode.<balanced(K,ps') | ps' <- split(K,ps)>
```

where `split(K,ps)` divides `ps` into `K` (or `#ps` if less) portions with sizes differing by at most one. This gives the nearest approximation to a `K`-branching tree that contains the same processes as a given list.

Each of the compression utilities is a function that takes a compression function (or any other function from processes to processes that is intended to leave the semantic value unchanged), a network in one of the two forms above, and the set of events that you wish to hide. So, for example,

```
LeafCompress(normal)(SysList)(H)
```

calculates the set of events from `H` that can be hidden in each `P(i)` (all that belong to `A(i)` and no other `A(j)`). If `H(i)` is this set of locally hidden events, the function then applies `normal` (the chosen compression operator) to each `A(i)\H(i)`, composes the rest in parallel, and then hides the events from `H` that belong to more than one of the `A(i)`.

The version of the dining philosophers from Section **??** provides a very good illustration of leaf compression. As we know, in analysing deadlock, we can hide all events. Each philosopher has three events that it does not synchronise with any process, namely `sits.i`, `eats.i` and `getsup.i`. It follows that these events get hidden and allow compression in `Leafcompress(normal)(S)(Events)` with `S` being either the symmetric or asymmetric version of the dining philosophers. In effect this reduces `Phil(i)` to the process

```
CP(i) = picksup.i.i -> picksup.i.(i+1)%N ->
        putsdown.i.(i+1)%N -> putsdown.i.i -> CP(i)
```

which can reasonably be regarded as the "essence" or skeleton of `Phil(i)` needed to examine deadlock. Compressing the `Fork(i)` processes has no effect, since all their events are synchronised. With 8 philosophers this reduces the asymmetric case's 1.6M states to 11,780. It is reasonable to expect good results from leaf compression whenever the individual processes have a significant number of actions (and this includes τ actions) that can be hidden and are not synchronised.

One can, of course, put parallel combinations of processes in a list rather than just sequential ones, and the same goes for the leaves of an `SCTree`. The obvious thing to do with the dining philosophers is to combine `Phil(i)` and `Fork(i)`, obtaining the list

```
ASPairList = <(PandF(i),APF(i)) | i <- <0..N-1>>

PandF(i) = ASPhil(i)[AlphaPhil(i)||AlphaFork(i)]Fork(i)
APF(i) =   union(AlphaPhil(i),AlphaFork(i)))
```

This, remarkably, reduces the 8-philosopher asymmetric check to 55 states, and the corresponding symmetric check finds the deadlock in its initial state (i.e. the compressed hidden system deadlocks before any actions have occurred).

The above methods, namely attacking individual processes or natural small combinations of processes usually works reliably. It is certainly worth experimenting with different compression functions and trying to formulate specifications so that as many events as possible can be hidden. The results obtained for the dining philosopers with this approach are exceptionally good, but we will see another much more ambitious and practical case where this applies in Chapter 2.

In some cases we might want to build up the network a process at a time, compressing as we go. The logical way to do this is to keep the size of the interface between the constructed parts of the system and the rest of the world as small as possible. Often this works well, but the following should be borne in mind:

- The time taken to do many large compressions can become significant.
- The size of system that FDR can compress is far less than the size it can handle on the right-hand side of a refinement check.
- As discussed earlier, a partial network may reach many more states when considered by itself as opposed to when it runs in the context of the whole network. Typically this is because the structure of the complete system imposes some invariant on the subnetwork. Therefore much of the work involved in compressing the partial system may be wasted.

A good example of the third point is a token ring such as that in Section **??**: the whole ring ensures that there is only one token present, but typically an open-ended section can allow any number of tokens in up to the number of nodes it contains. See Example **??** below and the practical entitled *A multi-dimensional puzzle* for

Figure 1.1 Inductive compression strategy applied to dining philosophers.

examples of how one can sometimes eliminate this type of problem by imposing the invariant on subnetworks.

The function `InductiveCompress`, applied to a list of processes, builds up the network in this way, starting from the end of the list. In other words, it compresses the two right-hand processes together, composes and compresses the result, and then iteratively adds on further processes, individually compressed, each time applying the chosen compression function. One must therefore arrange the list so that the processes are added in an appropriate order: in building up a linear network one will follow the natural order, meaning that only one process's communications that are internal to the network remain visible at any time. This is well illustrated by the dining philosophers. It is clear that we should arrange the processes so that they follow the natural order round the table, as indicated in Figure 1.1. The deadlock check is represented by

```
InductiveCompress(normal)(ASPairList)(Events)
```

This works exceedingly well, since each of the partially constructed lists with all but externally synchronised events reduces to only 4 states (with most being equivalent up to renaming). You can therefore check as many philosophers as you please on FDR, the only limit being the time it takes to compile the components. Leaving the `{|eats|}` or `{|getsup|}` events visible works similarly well, though the state space does grow with `N` if you leave `{|sits|}` visible since these events can prevent a philosopher being in contention with one of its neighbours.

Inductive compression can also work well on two-dimensional grids. One such example is provided by the file `matmul.csp` in the directory of examples relating to Chapter 13 of [?], which also contains some files with further one-dimensional examples. A further example can be found in the practical entitled *A class of puzzles.* In such cases it usually makes sense to start at a corner of the grid and add nodes in something like a "raster scan" order (namely adding all the nodes in one row one at a time before starting on the next row).

Example 1.5.1 Imposing an invariant This example illustrates how one can sometimes avoid the state space of one or more subnetworks, considered in isolation, expanding well beyond the limits that are imposed on them in the completed system.

In such cases, the whole network is imposing limits on the states the subnetwork can reach. Sometimes the user can identify what these limits are and express these *invariants* succinctly: in these cases one can often take advantage of them to prevent a compression strategy failing because of an explosion in the sizes of the compressed subcomponents.

A very simple `N`-node token ring is described by the processes

```
Nd(0) = b.0 -> c.0 -> a.1 -> a.0-> Nd(0)

Nd(r) = a.r -> b.r -> c.r -> a.(r+1)%N -> Nd(r)    (r>0)
```

so that `Nd(r)` can only perform `b.r` and `c.r` when it has been passed the token (`a.r`) but not yet passed it on (`a.(r+1)%N`). Initially `N(0)` has the token.

The model of the ring we would like to create is one in which the `{|a|}` events are hidden, and the others remain visible.

You will find that applying `InductiveCompress` to these processes works poorly, and compares very badly to the `2*N` states that the complete network has.

This is because the partially constructed networks can contain many tokens, but we can re-impose the invariant of there only being one as follows. The chain of processes `N(0)...N(r-1)` has "token passing" interface `{a.0,a.r}`. For each `r>0` we know that the token emerges from this chain before re-entering it, so that when the chain is put in the context of the one-token ring these events cycle `<a.r,a.0,a.r,a.0,...>`.

We could *enforce* this invariant on the chain by putting it in parallel with the process `EI(r)=a.r -> a.0 -> EI(r)`. We could add these enforcing processes (with alphabet `{a.0,a.r}`) to the network and they would not in fact change the behaviour of the complete network, but would avoid the compression of so many unvisited states.

The problem with this approach is that it builds in an assumption that our invariant holds without demonstrating that each of the chains does in fact remain within its constraints when the whole network runs. We can do better than this: the following process simply *assumes* that the invariant holds by saying that the process can behave however it likes if the invariant is violated.

```
AI(r) = a.r -> AI'(r) [] a.0 -> DIV
AI'(r) = a.0 -> AI(r) [] a.r -> DIV
```

Notice that, for any ✓-free process `P`, we have `P[|{a.0,a.r|}|]A(i) [FD= P` because `AI(r) [FD= RUN({a.0.a.r})`. This means that inserting these processes into the network creates an anti-refinement, and that it does not block any of the behaviour of the chains. It does, however, give the chains considerably less states for FDR to reason about in $\mathcal{N}$ since all that have more than one token present or less than none are divergent and largely ignored because of divergence strictness.

The network described:

```
<Node(N-1)>^<(AI(N-2-r),{a.0,a.N-1-r}),Node(N-2-r) | r <- <0..N-2>>
```

where the assumption processes are put in the appropriate places, therefore behaves well under inductive compression over $\mathcal{N}$. The fact that the complete network is divergence-free (unlike most of the processes constructed *en route*) in fact proves that the exception clauses of the `AI(r)` are never encountered during a complete run.

Note how we have used strict divergence to our advantage here. The reader will find that using a model without strict model does not work in the same way if its refinement-least member replaces `DIV` in the definition of `AI(r)`: while the overall model will still be accurate, there is no saving in states during inductive compression. A way of achieving a similar (though slightly less efficient) effect is to replace `DIV` by `RUN({error.r})` within `AI(r)`. This should prevent many states being explored following an `error.r` as it prevents the partial network from performing any further visible events. One should then ensure that no `error` event ever occurs in the complete system. See also Exercise **??** below.

The `SCTree` datatype referred to above has been introduced to make it easy for users to devise their own compression strategies, as well as to craft a hierarchy for a particular network:

```
StructuredCompress(compress)(t)(X)
```

builds up the network, first of all compressing all the leaves as in `LeafCompress`, and then combining each group of subnetworks implied by an `SCNode`. At every level of this tree except the topmost, as much hiding as possible is pushed to that level and compression applied. Thus the function from lists of process/alphabet pairs to `SCTree` defined

```
OneLevel(ps) = SCNode.<SCLeaf.p | p <- ps>
```

is such that applying `StructuredCompress(compress)` to the result is exactly equivalent to applying `Leafcompress(compress)` to `ps`.

We will find applications of this type in Chapters ???.

Compression on the left-hand side of a refinement check

It goes without saying that in cases where the process on the left-hand side of a refinement check has many states, compressions may also work well there. There are a number of special considerations in this.

The following two techniques can be used either in combination or separately:

- Use compression on proper sub-systems of the left-hand side but not the whole. In other words use the same compression strategy that you would on the right-hand side of a check. However in this case it is often a good idea to use `normal` as the compression function, since (for obvious reasons) the type of compressed structure it produces is unlikely to cause problems in normalisation.
- Use compression on the whole left-hand side: this may well be worthwhile since normalisation is a considerable transformation of the target process. In this case there is no point in using `normal`, so you should use `sbdia`, the composition of `diamond` and `sbisim`. Doing this may either increase or decrease the number of pre-normal form states FDR finds in the normalisation, but almost always makes normalisation faster when the left-hand side has a significant number of τ actions.

We presented the normalisation of the left-hand side, and the running of the refinement check, as two separate phases in Section **??**. This is how FDR normally performs them, but there is an option which can be advantageous in the case where the left-hand side has many more traces than the right-hand side. In that case FDR will have worked out the normal form behaviour of LHS on many traces where this is not necessary, perhaps at significant cost in terms of time. FDR defines a `transparent` function `lazynorm` that behaves like `normal` except that it only computes states of the normal form state machine that are actually explored in the context where it is used[6] If `lazynorm(P)` appears on the left-hand side of a check, then FDR relies on this function in its refinement phase of the check, so in effect normalisation and the main running phase are interleaved. This tends to be a disadvantage if RHS explores most of the traces of LHS, but can give a distinct advantage when it does not. This effect is illustrated in `pathol.csp`.

Exercise 1.5.1

Given a list `ps` of alphabetised processes, define functions that turn it into an `SCTree` such that applying `StructuredCompress` has the effects of (i) `InductiveCompress` on `ps`, (ii) groups the list into pairs of consecutive elements (and perhaps a singleton over) and composes and compresses these pairs, but applies no compression above this level, and (iii) divides up the list in to two pieces that differ in size by at most one, and applies inductive compression to each half, starting with the two processes at the ends of the original list – so we are inductively compressing to meet in the middle.

Exercise 1.5.2 Use FDR to discover a sequential process that it equivalent in $\mathcal{F}$ to

```
InductiveCompress(normal)(ASPairList)(diff(Events,{|eats|}))
```

for each `N>2`, where `ASPairList` represents the asymmetric dining philosophers as defined on page **??**. Note that this is bound to be a process that only uses the events `{|eats|}`. Explain the asymmetry in your answer.

Exercise 1.5.3 Modify both the definition of the token ring and the invariant assumption processes `IA(r)` from Example **??** so that there are always `1 <= K < N` tokens in the ring. What effect do you think the assumption process will have on the compression of the ring when `K=2`, `K=N/2` and `K=N-1`? [Note that if, when you perform a check of the complete ring with all assumptions in, it is not divergence-free, then your assumption processes must be wrong!]

Exercise 1.5.4 The non-$\mathcal{N}$ approach to a checkable "assumed" invariant set out at the end of Example ?? introduces a separate state with an error event that is reached when the invariant once the invariant *has been* breached. This is not as efficient in terms of the intermediate state spaces as the approach in which strictly interpreted `DIV` is reached, since the presence of divergence as soon as the invariant has been breached means that no further behaviour is examined, whereas `error.r` approach takes one more step.

Show that it is possible to improve on this by turning the event that breaks the invariant into an appropriate `error.r`. [Hint: use double renaming.]

[6]Also, `lazynorm` does not, because it cannot apply the strong bisimulation second stage present in `normal`.

1.6 Notes and reflections

FDR originally came into being in 1991 for use in a hardware design project, at the instigation of Geoff Barrett, then of inmos (see Note 2 below). Between that date and 2007 it was a product of Formal Systems (Europe) Ltd. Since 2008 it has been the responsibility of a team of Oxford University Computing Laboratory.

FDR underwent a major re-write in 1994/5, to become "FDR2", and from then on there has been a long series of incremental releases, of which the most recent one as this book was written is FDR2.91. That is the version which is described in this book. FDR2.91 is itself a major advance on FDR2.84 (the last release by Formal Systems). Hopefully the next year or two will see the release of FDR3, which we hope will be extended in various important directions, for example the ability to address its functionality other than through CSP and the incorporation of further algorithms for checking based on ideas such as SAT checking [**?**] and CEGAR **??**.

FDR's web site is ??? which potential users should consult for details of how to obtain it. At the time of writing, FDR is freely down-loadable for academic teaching and research, and most of the source code is also freely available for use by academic institutions, use of that code being restricted by a licence. Most of FDR is written in C++.

FDR is the work of many people. While the author has designed much of its functionality and some of the algorithms that make this possible, he has not written a single line of its code. Those most important to its development include Michael Goldsmith, who designed and wrote much of it and managed the project for many years; Bryan Scattergood, who designed and implemented the CSP_M input language that is so important to FDR's usability as well as the compiler; David Jackson and Paul Gardiner, who played vital parts in the development of FDR in the 1990's; and Philip Armstrong who both worked on it at Formal Systems for some years and now co-ordinates development of the code at OUCL.

Note 1: a lesson about Quicksort When FDR computes the strong bisimulation (`sbisim`) it works out a series of equivalence relations, each calculated from the previous one. The first relation is based purely on any marking of the nodes (as in a GLTS), and each subsequent one $\equiv_{n+1}$ is based on the marking plus the set of all pairs of the form (x, C) possessed by a node, where x is an action and C is a $\equiv_n$ equivalence class reachable under x from the node.

After performing this marking, FDR sorts a standardised representation of this information about the states so as to bring all nodes related under $\equiv_{n+1}$ together in the overall list. Since these lists may get quite long, it is important to use an efficient (i.e. *n log n*) sorting algorithm. It is well known that Quicksort has this as its expected time, with the chance of getting a significantly slower performance being vanishingly small if the pivot is chosen well (and picking the first element of the list as in the functional programming (page **??**) and CSP (page **??**) implementations in this does not qualify!) Imagine our surprise, therefore, when our first implementation, which used a library implementation of Quicksort, ran in what looked like deterministic n^2 time and was unacceptably slow.

We realised after a while what the cause of this was that, instead of splitting the list into three parts, `< x | x <- X, x<p >`, `< x | x <- X, x==p >` and `< x | x <- X, p<x >`, the library routine split it into two, `< x | x <- X, x<p>`, and `< x | x <- X, p<=x>`. When sorting a list of objects that were equal in the order (for example any set of states that are all equivalent under $\equiv_{n+1}$ in our application), this version of Quicksort takes *determinsitic* $O(n^2)$ time. Since there are, particularly for small n, often large numbers of equivalent states in our application, this explains why this library routine was completely unsuitable for FDR.

Note 2 inmos was an independent British company set up in the late 1970's to design and build microprocessors, and it was later taken over by SGS Thomson Microelectronics. Its main products were highly innovative. *Transputers* were the first microprocessors to have on-chip memory and communications hardware – creating a "computer-on-a-chip". Their communications hardware was designed, in essence, to implement a CSP-style handshake, and the architecture envisaged that not only would networks of transputers run in parallel, but that the programs that ran on a single transputer would often also be parallel, with the parallelism managed by fast context switching.

To support this concurrency, inmos introduced a language called OCCAM, itself a mixture of CSP and a simple imperative programming language. See Section **??** for more details of OCCAM. The design process of later generations of transputers involved a significant amount of formal specification and verification, based on CSP, OCCAM and other notations.

Barrett identified the need for a tool like FDR for the analysis of the relatively sophisticated communications hardware on a later transputer (the T9000???), and the basic algorithms of FDR were designed during a memorable meeting held in the summer of 1991 in the French town of Auch.

Reflections on technology and formal verification Perhaps the most striking thing about working with this tool for the last 18 years is the vast increase in the size of systems that it can handle now in comparison to then. Since the human mind has the same capacity to understand and solve problems as it did then, this means FDR (in common with some other automated formal methods) is useful on a much greater percentage of developments than it was then. Such tools were frequently over-sold in their early days, but have now become invaluable.

I recall how impressed I was when the first version of FDR handled a check of about 30,000 states in an hour or two, and that the following year running a check of about 600,000 states (the constrained solitaire board described in Section 15.1 of [?]) took about a day. By the time I published that book in 1997, the unconstrained solitaire board with 188M states was still out of range, whereas by 2000 and the revised edition it took 12-24 hours on a reasonably powerful workstation or 2.5 hours on a prototype (8 processor) parallel implementation. As I write this in early 2009 my powerful laptop will run the same check in 44 minutes and I am confident that by the time this book appears in print a parallel implementation will run it in a few minutes on an 8-core PC. And this is in running the same model, visiting the same set of states once each using (at least since the mid-1990's) little-changed algorithms and code. An experimental way of finding FDR's counter-examples quickly using *bounded model checking* based on SAT checking – a completely different way of examining the state space implemented by Hristina Palikareva – has already been able to solve the unconstrained board in under 4 minutes [?] on a single core. Presumably, and hopefully, this progress will continue.

While programmers and algorithm designers might like to take credit for the vast increase in capability since then, in truth most of the advance in power is due to the continuing application of Moore's law both to the processing power and memory size of computers, as illustrated above. All we have had to do is find ways of using this, and in particular the recent phenomenon by which the increase in single-CPU speed has been replaced by an increase in the number of cores.

This is not to say that advanced techniques such as compression and the SAT-checking advance described above do not sometimes have a huge effect on what examples can be approached, and we will see many in this book. It is just that in practice the effect of these is only rarely on a par with the spectacular basic computer advances detailed above.

The following is an atomic equivalent program for the original Bakery Algorithm with `Maximum A`.

Chapter 2

Shared variable programs

This is the first of two chapters devoted to the topic of *shared variable* programming. They can be viewed in several ways: as a substantial case study in writing practical CSP, as a lesson in how to build simulators and compilers in CSP for another notation, and guide to a new tool for assisting in analysing shared variable programs.

In this chapter we will build a CSP model of how to run a shared variable program. Since this model takes the form of a compiler written in CSP it enables us to analyse the behaviour of shared variable systems; we we show how FDR's compression functions work particularly well on the resulting CSP processes and how they are thus integrated into the compiler; and we give two substantial case studies. Finally, we describe how the compiler has been given a front end GUI, giving rise to a tool called *SVA*. In Chapter **??** we will examine the semantics of shared variable programs in more detail, and describe various advanced features of the compiler, illustrating them by expanding on our case studies.

In these two chapters we will consider programs that consist of a number of *thread* processes that do not communicate directly, but rather interact by writing to and reading from variables that they share.

We will study a simple shared variable language in which each thread is written in a small sequential language, and the shared variables are of either integer or Boolean type, including one-dimensional arrays. For the time being[1] we will assume that all variables and constants are declared *and initialised* at the outermost level (i.e. for all parallel threads): this is clearly necessary for all shared variables, and adopting this uniform approach makes compilation easier.

[1]We will be studying two forms of the language: first a somewhat unconventional one that is closer to implementation and then a more conventional one that is what most users will actually write. We will allow locally defined variables in the latter.

Here are two types of thread process that are intended to implement mutual exclusion: in each case we run two processes in parallel with $i \in \{1,2\}$, and both are supposed to ensure that the two critical sections never run at the same time. Mutual exclusion is an important aspect of practical concurrent programming, since it is used to prevent processes from accessing resources simultaneously when this could lead to an error.

```
H(i) = {iter {b[i] := true;
             while !(t = i) do
               {while b[3-i] do skip;
                t := i};
             {CRITICAL SECTION}
             b[i] := false;}}

P(i) = {iter {b[i] := true;
             t := 3-i;
             while b[3-i] && (t = 3-i) do skip;
             {CRITICAL SECTION}
             b[i] := false;}}
```

H stands for Hyman's algorithm [] and *P* stands for Peterson's algorithm [**?**]: both were proposed as solutions to the mutual exclusion problem. In both algorithms, `t` and `i` are integers (with `i` being treated as a constant since it is the parameter) and `b` is an array of two Booleans. In both, `b[i]` is a flag that process `i` sets to indicate that it wants to acquire a critical section, and they use these and the variable `t` to negotiate who goes first in the case that processes 1 and 2 both want to perform a critical section.

The only feature of the above programs that might be unfamiliar is `iter`, which an abbreviation for `while true do`.

In trying to understand how these two algorithms work in practice it is important to realise that the various actions of the two threads are *arbitrarily interleaved.* A thread may have to access several variables during an expression evaluation: we assume that accesses by the other process might well occur between these, and between an expression being evaluated and the value being used. Our language also contains a construct `atomic C` that ensures that the execution of the command `C` is not interleaved with any actions by another thread. Not only has `atomic` not been used in the above thread, but mutual exclusion is needed in practice to implement atomic execution: it follows that one should not normally use `atomic` in implementing it!

In fact, only one of the two algorithms above succeeds in guaranteeing mutual exclusion. See if you can work out which.[2]

Mutual exclusion is a fundamental building block of concurrent programming, and indeed many practical implementations of CSP-style handshaking depend on it. We will see both further mutual exclusion algorithms and applications of them in the case studies and exercises later in this Chapter.

The rest of this chapter is devoted to a compiler called SVA (shared variable analyser) for shared variable programs of this sort, that is actually written in CSP. It is primarily intended as a case study of how to compile other languages into CSP, but also provides interesting lessons on how to write CSP and use FDR's compression functions effectively. The author also hopes that SVA is useful in its own right.

SVA, with a front end that provides a parser, GUI and counter-example interpreter, is available from this book's website. Before going further in this chapter (but after applying some thought to the two mutual exclusion algorithms above) the reader is advised to try both the raw compiler (by running FDR on the files `hyman.csp` and `peterson.csp`) and SVA (by checking the two corresponding `.svl` files). Notice in particular the far clearer debugging output from SVA.

2.1 Writing a compiler in CSP$_M$

CSP, and even CSP$_M$, which is extended by functional programming, seems a most unlikely language to write compilers in. It has not until very recently (after SVA was written) had a type of strings, it has no conventional `write` or other command to save the result of compilation into a file, and it certainly was not designed with this style of application in mind.

The compiler we will be describing overcomes the first of these objections by starting from a "source" program written as a member of a CSP data type. Such programs can be written by the user, or more usually created from an ASCII syntax for the language by the parser in SVA.

Our compiler does not have to output the resulting CSP in the conventional sense, since it is the combination of it and the source file (itself in CSP syntax) that is input to FDR. It would perhaps be more accurate to call it a simulator, since what it does is to build a CSP model that behaves as the source program does. It is, however, fairly accurate to think of the combination of it and the CSP compiler within FDR as a true compiler from the source language to FDR's internal notations.

[2]This was an exercise set for a number of years by Oxford lecturers. This exercise inspired the creation of the compiler described in the present chapter.

The compiler described in this section is the one in `svacomp.csp`, simplified to remove the support for a few advanced features that we describe in the next chapter.

2.1.1 Data types

The compiler is a CSP program `svacomp.csp` that is `include`d in the source file, and which itself `include`s the standard compression utilities `compression09.csp`.

To use the compiler, the source program must define various constants which we will introduce gradually. First, we will concentrate on how to describe a thread process and how the compiler deals with these.

There are three types that are fundamental to the description of a thread process: those of *Commands* `Cmd`, *Integer Expressions* `IExpr` and *Boolean Expressions* `BExpr`.

The integer expressions are as follows:

```
datatype BinIOps = Plus | Times | Minus | Div | Mod | Max | Min
datatype UIOps = Uminus

datatype IExpr =  IVar.ivnames | IArc.(ianames,IExpr) |
                  Const.{MinI..MaxI} | BIOp.BinIOps.IExpr.IExpr |
                  UIOp.UIOps.IExpr | ErrorI
```

In other words, an integer expression is formed from integer variables, array components and constants using binary and unary operations, and for technical reasons that will be explained later, we need special syntax denoting an error. The Boolean expressions are similar:

```
datatype BinBOps = And | Or | Xor
datatype CompOps = Eq | Neq | Gt | Ge | Lt | Le

datatype BExpr = BVar.bvnames | BArc.(banames,IExpr) |
                 True | False | Not.BExpr |
                 BBOp.BinBOps.BExpr.BExpr |
                 CompOp.CompOps.IExpr.IExpr | ErrorB
```

In this syntax there is an asymmetry because Boolean expressions may include integer ones, but not vice-versa.

The syntax of commands is as follows:

```
datatype Cmd = Skip |
                Sq.(Cmd,Cmd) | SQ.Seq(Cmd) |
                Iter.Cmd | While.(BExpr,Cmd) |
                Cond.(BExpr,Cmd,Cmd) |
                Iassign.(IExpr,IExpr) | Bassign.(BExpr,BExpr)|
                Sig.Signals | ISig.(ISignals,IExpr) |
                Atomic.Cmd  | ErrorC
```

Here, `Skip` is a command that does nothing, `Sq` and `SQ` are respectively binary and list forms of sequential composition, `Iter` runs its argument over and over for ever and `While.(b,C)` evaluates `b` and then runs `C` followed by the loop again, or `Skip`. `Cond(b,C,D)` is a standard conditional equivalent to `if b then C else D`. There are separate commands for integer and Boolean assignment, each evaluating their second argument, determining what location the first represents, and performing the assignment. `Sig.s` and `ISig.(sc,e)` both cause the thread process to send a CSP-like communication to the environment; the first is the simple event `s`, the second outputting the value of expression `e` along the integer channel `sc`. These two constructs are included in the language to make specification easier: there is no mechanism for synchronising the signals produced by different threads, and in fact the signals used by different threads must be distinct. `Atomic.C` is as described earlier, and `ErrorC` is again included for technical reasons.

The data types above are the main ones that those that someone writing a program that uses `svacomp.csp` directly needs to understand, since they are the ones in which programs are written.

2.1.2 Variable names

In contrast to the complex types above, the types used within the programs are simple: just integer and Boolean. Because FDR only supports finite state component processes, and each variable process we will use has a state for every value it might take, in practice we can only handle programs using comparatively small ranges of integers – how small depends on the rest of the program – and two important constants that have to be defined in each program are `MinI` and `MaxI`. Each integer variable or array is declared to contain members of any subtype of `{MinI..MaxI}` and the index type of any array is also a subtype. Each program contains a number of constants that allow the compiler to compute, for each array name `a`, its index type `itype(a)` and, for each integer variable or array component `v`, its content type `ctype(v)`. These types are always subsets of `{MinI..MaxI}`.

The name of each variable or array component used in programs is drawn from the following datatype:

```
datatype namestype = IV.Int | IA.Int.Int
```

```
                    | BV.Int | BA.Int.Int | NonVar
```

Thus each integer variable takes the form `IV.n`, each integer array has the form `IA.m` where the `j`th component (starting from `0`) is `IA.m.j`, and similarly for Booleans.[3]

The introduction of a string type into CSP$_M$ means that the `Int` in each of these clauses can be replaced by `String` in future versions of SVA. The principal advantage of this would be making the debugger output directly from FDR more readable.

`NonVar` is a constant used to handle errors.

Each program using `svacomp.csp` must define sets `ivnums`, `ianums`, `bvnums` and `banums`: the variable and array numbers it uses. We can now define sets of the arrays on one hand and the variables and array components on the other, that are used in the program:

```
ianames = {IA.j | j <- ianums}
banames = {BA.j | j <- banums}

ivnames = union({IV.j | j <- ivnums},
                {IA.j.k | j <- ianums, k <- itype(IA.j)})
bvnames = union({BV.j | j <- bvnums},
                {BA.j.k | j <- banums, k <- itype(BA.j)})
```

Notice that these are used in the definitions of `IExpr` and `BExpr`.

In the rest of this chapter we will refer to members of these sets as *locations*, namely slots in the machine store where values are stored.

2.1.3 Compilation strategy

A program consists of the declarations and initialisations of the variables and arrays it uses, together with a collection of thread processes (members of `Cmd`) that use them.

We build a network that consists of one process for each thread, and (usually) one process for each location. The thread processes do not communicate directly with each other at all. They just read and write the variable processes and communicate externally via their signals and, if they need to start and then end an atomic section, communicate with the processes that govern these.

An individual location (i.e. variable or array component) is a very simple process, particularly in the absence of any `atomic` constructs:

[3]The SVA front end allows more conventional names for variables, and builds up a table associating each such name with one derived from `namestype`.

```
IVAR(x,v) =
          let t = ctype(x) within
            (iveval?_!x!v -> IVAR(x,v)
         [] ivwrite?_!x?w -> IVAR(x,w)

BVAR(x,v) = bveval?_!x!v -> BVAR(x,v)
         [] bvwrite?_!x?w -> BVAR(x,w)
```

Each thread will be allocated a distinct index, and the first parameter of `ivwrite` etc is the thread that the variable is being accessed by. The second is the name of the location (variable or array component) and the final one is the value being read or written.

When a program does include atomic execution, the variable processes help implement this. While a thread runs atomically, we require that no other thread accesses variables, or itself enters an atomic section. Both these requirements are met by modified variable processes. We give only the Boolean version here.

```
BVAR_at(x,v) = bveval?_!x!v -> BVAR_at(x,v)
            [] bvwrite?_!x?w -> BVAR_at(x,w)
            [] start_at?j -> BVAR_inat(j,x,v)

BVAR_inat(j,x,v) = bveval.j!x!v -> BVAR_inat(j,x,v)
                [] bvwrite.j!x?w -> BVAR_inat(j,x,w)
                [] end_at.j -> BVAR_at(x,v)
```

In other words once some process enters an atomic section, only it is allowed to read and write the location until the atomic section ends, and no other process can enter an atomic section during this period.

An atomic section is uninterrupted; it behaves as though it all happens in an instant as far as the other threads are concerned. That last analogy will be very useful to us later (Section **??**, where we consider refinement between shared variable programs), but it breaks down when more than one signal event can occur during a single atomic sections. We will therefore regard it as good practice to ensure that no atomic section can output two or more signals; the results of that section are restricted to that case.

2.1.4 Compiling a thread

Apart from the evaluation of expressions, the reduction of a thread process into CSP is remarkably straightforward. The complete translation for the `Cmd` type is given below. The first few clauses do not involve evaluating expressions:

```
MainProc(Skip,j) = SKIP

MainProc(Sig.x,j) = x -> SKIP

MainProc(Sq.(p,q),j) = MainProc(p,j);MainProc(q,j)
MainProc(SQ.<>,j) = SKIP
MainProc(SQ.<p>^Ps,j) = MainProc(p,j);MainProc(SQ.Ps,j)

MainProc(Iter.p,j) = MainProc(p,j);MainProc(Iter.p,j)
```

Expression evaluation is incorporated into the following definitions in *continuation* style: to calculate the value of an expression as part of a program, we use functions `BExpEval(b,P,j)` and `IExpEval(e,P,j)` where

- `b` and `e` are the expressions to be evaluated,
- `P` is a continuation: a function that expects a Boolean or integer as appropriate, and behaves like the rest of the program after it has been given such a value.
- `j` is the index of the thread. This is included so that any error signal that occurs during the expression evaluation can be labelled properly.

So the local definition in the following pair of constructs defines a continuation which expects a Boolean, and the controlling expression `b` is then evaluated with this continuation.

```
MainProc(While.(b,p),j) =
        let P(x) = if x then MainProc(p,j);MainProc(While.(b,p),j)
                        else SKIP
        within BExpEval(b,P,j)

MainProc(Cond.(b,p,q),j) =
        let P(x) = if x then MainProc(p,j) else MainProc(q,j)
        within BExpEval(b,P,j)
```

The effect of a Boolean write statement is simply to generate a communication `bvwrite.j.lv.rv` where `j` is the thread index, `lv` is the location to which the write will be made, and `rv` is the value to be written. Notice that these cases make use of additional functions that supply continuations with respectively a Boolean and integer *location* when applied to a suitably structured expression. Thus, below, `BLvEval(el,Q,j)` is a function that evaluates the `BExpr el` as a Boolean *left* *value*: a Boolean location that naturally sits on the *left* of an assignment. To evaluate successfully in this way the expression either has to be a variable or an array component `a[e]` where `e` is an integer-valued expression.

```
MainProc(Bassign.(el,e),j) =
                          let Q(lv) =
                          let P(rv) = bvwrite.j.lv.rv -> SKIP
                              within BExpEval(e,P,j)
                          within BLvEval(el,Q,j)

MainProc(Iassign.(el,e),j) =
                          let Q(lv) =
                          let P(rv) = if member(rv,ctype(lv)) then
                                              ivwrite.j.lv.rv -> SKIP
                                      else error.j -> STOP
                              within IExpEval(e,P,j)
                          within ILvEval(el,Q,j)
```

The effect of these definitions is to do the calculations necessary for working out the location (or *left value*) `lv` are done before those for the assigned value (or *right value*) `rv`. It is straightforward to achieve the opposite order (see Exercise **??**). Which order is chosen makes a real difference, because it alters the order in which variables are read.

For integer locations, we need to check that the integer written is within the correct range. If not a run-time error occurs.

The `Eval` functions can generate an execution error, for example through division by zero or array index out of bounds. In such cases these functions just generate an appropriate error message and stop: the continuation argument is not used.

The final three clauses of the definition of `MainProc` are straightforward given what we have already seen.

```
MainProc(ISig.(c,e),j) = let P(x) = c!x  -> SKIP
                          within IExpEval(e,P,j)

MainProc(Atomic.p,j) = start_at.j -> MainProc(p,j);end_at.j -> SKIP

MainProc(ErrorC,j) = error.j -> STOP
```

2.1.5 Evaluating an expression

The evaluation methods for integer and Boolean expressions are very similar. Because integer expressions are self contained, we will concentrate on these here and define `IExpEval`. `svacomp.csp` evaluates an expression in two stages: first, it instantiates any free variables and array components in the expression one at a time until there are none left; then it reduces the resulting expression to a single value.

Because of the way we are structuring our model, each fetch from a location generates a communication of the form `iveval.j.v?x`, where `j` is the index of the process and `v` is the name of a variable or array component.

The order in which these fetches are performed is important, since other threads may be writing to the locations used at the same time. The way in which `svacomp.csp` orders them is left-to-right, subject to any expression representing an array index needing to be evaluated in full before the array component can be fetched. This order is implemented through the function `ffetchi(e)` whose definition, together with that of the datatype `fetch` it returns, is given below.

```
datatype fetch =  NoF | ISV.ivnames | BSV.bvnames | ErrorX

ffetchi(IVar.n) = ISV.n
ffetchi(IArc.(v,e)) = let f = ffetchi(e)
                 within
                 (if f == NoF then
                    (let k=evaluatei(e) within
                       if ok(k) then
                         let n=num(k) within
                           (if member(n,itype(v)) then ISV.v.n
                                                  else ErrorX)
                       else ErrorX)
                  else f)

ffetchi(Const.k) = NoF

ffetchi(BIOp._.ea.eb) = let ffe = ffetchi(ea) within
                     if ffe==NoF then ffetchi(eb)
                                 else ffe

ffetchi(UIOp._.e) = ffetchi(e)

ffetchi(ErrorI) = NoF
```

The type `fetch` gives the result of a search for the first fetch required to evaluate an expression. `NoF` means that no fetch is required: the variables (including array components that appear in the expression) are fully instantiated. `ISV.v` or `BSV.v` denote an integer or Boolean location.

The function that actually evaluates an expression is then

```
IExpEval(e,P,j) =
          let IXEF(NoF) =
                 let k=evaluatei(e) within
                    if ok(k) and num(k)>=MinI and num(k) <=MaxI then
                             P(num(k))
                       else error.j -> STOP
              IXEF(ISV.v) = iveval.j.v?x -> IExpEval(subsi(v,x,e),P,j)
              IXEF(_) = error.j -> STOP
          within IXEF(ffetchi(e))
```

The two additional functions used here are `evaluatei` which turns a fully evaluated integer expression into an integer, and `subsi`, which substitutes the integer value `v` for location `x` in the integer expression `e`. We omit these (long but straightforward) definitions here: the reader can find them in `svacomp.csp`.

We now know how to compile a thread process, and how to create a process modelling a location. These are the only components of our model of a program. In the case where there are no `Atomic` constructs, an accurate simulation can be produced by interleaving (`|||`) all of the thread processes, and interleaving all the location processes, and synchronising these two collections on the events `{|iwrite,bwrite,ieval,beval|}`.

It should be clear that any `Atomic` construct nested inside another has no effect on the intended execution of processes. On the other hand such constructs make compiler writing more difficult and add extra actions into the CSP simulation. `svacomp.csp` therefore removes all nested `Atomic`s using the function `One_at` before compilation (only outermost ones remain).

If we put the system together more carefully we can, however, gain considerable scope for applying FDR's compression operators, and that is the subject of the next section.

Pros and cons of compiler writing in CSP

There are clearly three languages involved in any compiler: the *source* one that is being compiled, the *target* language, and the one in which the compiler itself is written.

The inability of CSP$_M$ to handle strings at the time of its creation meant not only that the compiler has to input programs represented in CSP data-types such as `Cmd`, but that it is incapable of outputting the text of the CSP program that is the result of the translation. Therefore, the compiler itself has to interpret the source program as a CSP program each time we want to "run" the former.

This seems to work well in conjunction with FDR, but it would be unsatisfactory if we wanted to do inspect the CSP result of the translation, or to input it into a tool that did not have a full understanding of the functional part of CSP$_M$.

It would be straightforward to adapt `svacomp.csp` so that instead of generating actual process behaviour, as it does now, it created a representation of a CSP process in a syntactic datatype analogous to `Cmd`. Thus, for example, we might write

```
SMainProc(Atomic.p,y) = SSeq.((SPref.start_at.j,SMainProc(P,j)),
                              (SPref.end_at.j,SSKIP))
```

rather than the definition on page **??**, where `SSeq`, `SPref` and `SSkip` are the constructors representing the obvious CSP syntax.

We might notice that the result of the translation would be a fully laid-out CSP program, with no use of the functional constructs in CSP$_M$. So, for example, each parallel process would have a separate existence in the syntax.

Our compiler would then be a pure functional program: it would not communicate itself at all. That version could be translated very easily into functional languages such as Haskell.

We conclude that it is very much a matter of personal taste, when compiling a language into CSP, whether to write the translation in CSP or in some other language. The author has tended to follow the former approach [**?**], whereas others have tended to follow the latter [**?**, **?**]. The main reasons why we have used the former in this chapter are firstly that it provides an excellent illustration and case study of just how expressive the CSP$_M$ language is, and secondly that readers of this book should (at least by this point) be familiar with CSP$_M$ but may not be familiar with Haskell.

2.2 Applying compression

The simple way of composing the system model as `Threads [|X|] Locations` leaves all synchronising until the very last moment. This is exactly like interleaving all the philosophers and all the forks in the dining philosophers (see Section+**??**+), and, like that example, leaves no prospect for compression other than any that might be available on the individual nodes.

Just as in that example, we must to aim to get many of the interactions between the processes established early in the composition tree of our network. The main technique that `svacomp.csp` uses to achieve this is to associate each location process with a specific thread. It therefore has to choose *which* thread. There are two levels of priority for this:

- If the user specifies that some location will go with a given thread, it does. To achieve this, when a thread is compiled, it is coupled with a pair of sets: the integer and Boolean locations that the user wishes to attach to it. Thus our descriptions of networks are built not from `Cmds` but from `Threads`, a type synonym for `(Cmd,(Set(ivnames),Set(bvnames))`.
- If the above does not apply, then `svacomp.csp` chooses a thread automatically.

We will concentrate on the final case. To make a sensible choice, we need to work out which threads use a given location for reading, and for writing. In the case of ordinary variables this is simple textual analysis, but things are harder in the case of array components.

While one can imagine more sophisticated analysis, what `svacomp.csp` does is to work out which array indexing expressions are actually constant because they contain no variable or array component. This is common because, just as in the Hyman and Peterson cases above, many examples have a parameterised family of threads, in which the parameter (`i` in the cases mentioned) is treated as a constant (`Const.i`), since `i` has a specific integer value for each thread. It reduces these expressions to their values and replaces (for example) `IArc.(IA.1,BBOp.Plus.Const.2.Const.1)` by `IVar.IA.1.3`. Where an array index is not constant, the thread is assumed to be able to access any component.

The function `ConstC` is applied to each thread before compilation to carry out this reduction of constant expressions, supported by `CBEx(b)` and `CIEx(e)` that respectively determine whether a Boolean or integer expression is constant.

It is this reduction that accounts for many of the error elements in types. For the reduction may produce an array index out-of-bounds or other "run time" error, so the result is an error expression or `Cmd` as appropriate.

Similarly, when evaluating an integer or Boolean expression, the result type is one of

```
datatype EInt = Ok.AllInts | NotOk
datatype ExtBool = BOk.Bool | BNotOk
```

The `NotOk` elements in these types permit errors to propagate smoothly: this allows such errors to cause run-time error events in the compiled program, rather than compilation errors in FDR that stop the program being modelled.

The decisions about which locations are associated with which threads are made by a function `Analyse`, which is applied to a list of processes coupled with user-declared sets of local variables and returns many pieces of information about each thread in the context of the network:

- `IW, IR, BW` and `BR` give the integer and Boolean locations each thread can write and read (most of the returned parameters are, like these, functions from the thread's index to its value).
- `LIV` and `LBV` give the integer and Boolean locations that are to be grouped with each thread.
- `ROI` and `ROB` give the variables that each thread can read and *nobody* can write.
- `Alpha` and `Alpha_at` give, in the cases without and with `Atomic` constructs, the alphabets of each thread/location combination when all of these are combined in parallel.
- `LI` gives the interface between each thread and its own associated locations.
- `SEv` gives the set of signal events used by each thread. These sets of signals must be disjoint with each other.
- `UseAtomic` is a flag that is true when any of the threads have an `Atomic` construct.

The automatic allocation of locations to threads (i.e. where no over-riding allocation applies) is done as follows:

- If at least one thread is able to write to the location, then such a thread with the lowest index is chosen. In the common case where only one thread can write to a location, this means that all writes to that variable can be hidden immediately.
- If no thread can write to the location, then that location has a constant value through each run. There are a number of ways in which we could optimise this; what `svacomp.csp` does is to give each thread that can read the location a separate read-only variable process, meaning that all reads of that variable are hidden immediately.[4]

[4]A perhaps superior approach would be to transform each thread so that every read of the "non-variable" becomes a constant equal to its initial value, though this might not be possible in the case that an array with different constant values in its various components is read-only, since

The author's experience is that this automatic allocation usually works very well except in large examples where a number of locations are writable by many or all threads. The issue that arises there is that many shared variables will now be allocated to a single thread (one with low or least index). This is not a problem in itself, but in some cases the combination of a single thread process with many variables that others can write[5] will have many states, and this might cause problems at the compression stage. If this happens it is a good idea to distribute multiply-writable locations by hand.

The function `CompileList` turns lists of threads into a list of CSP processes. Each of these is the parallel combination of `MainProc` applied to the `Cmd` representing the thread, and the processes implementing this thread's local variables, with the read and write actions linking these hidden. It chooses two options depending on whether `UseAtomic` is true or not. If not, then (except for the peculiarities relating to read-only locations) the result is just a re-arrangement of the `Threads[|X|]Locations` network discussed earlier.

If `UseAtomic` is true, then the versions of locations designed for atomic execution are used, and in addition each thread process is put in parallel with a process `At_Reg(i)` whose job is to prevent the thread from performing any signal actions during another thread's atomic section.

`svacomp.csp` provides a number of different functions for structuring and compressing these networks. The most basic is one that simply puts all the processes in parallel without any attempt at compression.

```
Compile(C) =  ListPar(CompileList(C))
              \({|iveval,bveval,bvwrite,ivwrite,start_at,end_at|})
```

This is provided mainly as a point of comparison so that the user can judge how effective compression is. The simplest (and often best) compression strategy is to apply Leaf compression (see Section **??**). It simply compresses the combinations of threads and associated locations, hiding the same internal actions as above:

```
CompressedCompile(C)(AS) =
   LeafCompress(compress)(CompileList(C))(diff(Events,AS))
```

The set `AS` is the alphabet of the specification: the events that need to be left visible to determine whether or not the process satisfies its specification. This will

one would not know which constant value a particular read is until run-time. This last case would essentially be a look-up table.

[5]This is a problem since, when looking at this combination, the external writes mean that every state of the thread will be multiplied by the product of the sizes of the types of all such locations, since there is nothing to stop external states making arbitrary writes to these at any time.

almost always be a set consisting of error events and signal events. In this context `Errors` means the set of all run-time errors `{|error, verror|}`, so that one way of checking to see if any such error can occur is

```
assert STOP [T= CompressedCompile(System)(Errors)
```

The best way of allowing more ambitious compression strategies is via networks presented hierarchically as trees. A basic network is a single thread, and we can compose any number of networks into a single one. The datatype that describes this is

```
datatype CmdStruct = CSLeaf.Thread | CSNode.List(CmdStruct)
```

This is designed to support the following strategy:

- Each thread process is compiled and grouped with its associated locations exactly as above.
- These are indvidually compressed as in `LeafCompress` above.
- Except at the root of the tree, the subnetworks below each `CSNode` are composed in parallel, all internal commnications hidden, and compressed.
- The list of subnetworks at the root of the tree are put in parallel and have internal communications hidden, but not compressed because (see Section **??**) it is productive to compress at the top level of a strategy.

Thus the following `CmdStruct` will give a compression strategy that is identical to `leafcompress`: `CSNode.<CSLeaf.C | C <- ThreadList>` for a list, `ThreadList` of `Threads`.

The function that `svacomp.csp` provides to perform this hierarchically compressed compilation is

```
HierarchCompressedCompile(ct)(AS) = PSStructCompile(compress)
                                      (CompiledStructure(ct))
                                      (diff(Events,AS))
```

Here, `CompiledStructure` is a function that behaves exactly like `CompileList` except that instead of turning a list of `Cmds` into a list of processes, it turns a `CmdStruct` into a `SCTree`, the type of trees of processes described on page **??**, naturally preserving the tree structure. We can therefore use the function `PSStructCompile` (from `compression09` and described in the same place) to perform the compression. `AS` is again the set of events (almost always consisting of error and signal events) that we wish to be visible for checking against some specification.

The implementation of `CompiledStructure` in `svacomp.csp` is straightforward: it flattens out the `CmdStruct` into a list, applies `CompileList`, and structures the result into a `SCTree` using the original argument as a model. One can imagine more sophisticated versions in which the allocation of locations to threads takes account of the tree structure: if there is a choice, then allocate a location with multiple writers to one of these in a subnetwork where as many other threads use the location as possible.

One can also imagine a function that takes a list of threads and arranges these into a `CmdStruct` in such a way that as many interactions as possible will be hidden early.

2.3 The front end of SVA

When the author wrote the first version of the shared variable compiler in 2000, writing programs as literal members of `Cmd` was the only way of using it. It became much easier to use when David Hopkins created a front end and GUI for it in 2007, together with a parser that converts an ASCII form of the language into `Cmd` etc. He has been kind enough to update this front end to encompass the much-extended functionality of the version of the compiler described in this chapter.

2.3.1 SVL

We have devised an ASCII syntax for the shared variable language the tool uses, tentatively called SVL. This is deliberately simple: we intend that can be used to describe simple algorithms rather than large systems.

A *program* consists of global declarations and initialisations, and a list of *threads*. Each thread consists of an optional *claim* of shared variables, so that the user can guide the system about which shared variable to group with which thread (as described on page **??**), optional declarations of *local* variables (ones not readable by any other thread) and a *process* that represents a member of `Cmd`.

We will add further feature to the language later in this chapter. An example program is given below.

This example illustrates the syntax that corresponds to each of the features of `Cmd` and expressions described in Section **??**. A full syntax can be found at `web site???`

2.4 Specifications in SVA

SVA provides two ways of creating specifications that FDR can check of programs in SVL. The first of these is to build a CSP specification that refers to the signal events that may be communicated by the program. So for example we can, in a mutual exclusion example, make each thread's critical section `sig(css.i);sig(cse.i)` and use the CSP specification `MESpec = css?i -> cse!i -> MESpec`. This states, as a trace specification, that each "critical section start" `css.i` is followed by the corresponding "criticial section end" `cse.i` before any other critical section can start.

More simply yet we can easily check that some error signal does not happen, and in most cases we will want to check in this way that no run-time error signal can occur: the

2.5 Case study: Lamport's Bakery Algorithm

The Bakery algorithm was designed by Leslie Lamport [] as a mutual exclusion algorithm that works between an arbitrary number of threads all potentially needing to perform critical sections that must exclude all the others. It works roughly on the principle by which some shops have a machine dispensing tickets that allocate "turns" to customers: the tickets are marked with ever increasing numbers on and there is typically a display that shows what ticket is now being served.

There are, however, some significant differences between the shop algorithm and Lamport's version. The former clearly has a single server handing out tickets, but in the Bakery Algorithm each process computes its own ticket value. These return to 0 each time there is no-one waiting. The distributed and unsynchronised nature of the Bakery Algorithm means that tickets are not necessarily issued in sequential order and may indeed take equal values. To resolve the latter problem the nodes are given an index that is used to give priority in the case that two ticket values are equal. In the following, $(a, b) < (c, d)$ if $a < c$ or $a = c$ and $b < d$.

The following description of the algorithm is taken from Lamport's original paper, modified only by turning the 0/1 integer array *choosing* into a Boolean array, and converting the **goto** commands into an equivalent control structure in our own language.

```
iter
  begin integer $j$;
    $choosing[i]$ := true;
    $number[i]$ := $1 + maximum\{number[1], \ldots, number[N]\}$;     $(*)$
    $choosing[i]$ := false;
```

```
        for $j = 1$ step $1$ until $N$ do
            begin
                while $choosing[j]$ do skip;
                while $number[j] > 0$ and $((number[j], j) < (number[i], i))$ do skip;
            end;
        $critical\ section$;
        $number[i] := 0$;
    end
```

Original Bakery Algorithm

Thus thread i first sets the Boolean *choosing*(i) to *true* so that any other thread can see that it has started to seek to acquire the mutual exclusion "token" but has not yet selected its ticket value. Then it selects its ticket value as one greater than the maximum of the values it reads for all the *number*(j)s. Lamport emphasises that this is not intended to be an atomic calculation.

Having selected its ticket value, thread i looks at the variables of each other thread j (the loop also looks at $j = i$, but that is unnecessary since both of the subloops will terminate immediately). For each j it waits until thread j is not choosing and either has a greater ticket value or an equal one when $j > i$. When this is done for all j our node can assume it has claimed the notional token and can perform its critical section. When it is finished it simply sets *number*$[i]$ to 0.

This seems straightforward until one realises that the various nodes are potentially changing their variables at the same time at arbitrary rates relative to one another. For example two threads might be calculating their maxima at essentially the same time, but need not see the same values from the other threads.

In many other presentations of this algorithm, for example [] the roles of *choosing* and *number* are combined into a single array, say *turn*, with the approximate relationship

- $turn[i] = 0$ corresponds to $number[i] = 0$ and $choosing[i] = false$;
- $turn[i] = 1$ corresponds to $number[i] = 0$ and $choosing[i] = true$;
- $turn[i] > 1$ corresponds to $number[i] = turn[i] - 1$ and $choosing[i] = false$;

```
iter
    begin integer $j$;
        $turn[i] := 1$;
        $turn[i] := 1 + maximum\{turn[1], \ldots, turn[N]\}$;    $(*)$
        for $j = 1$ step $1$ until $N$ do
            **begin**
                **while** $turn[j] > 0$ **and** $((turn[j], j) < (turn[i], i))$ **do skip**;
            **end**;
        *critical section*;
        $turn[i] := 0$;
    **end**
```

Simplified Bakery Algorithm

The above programs translate naturally into SVL except for the N-way *maximum*, which therefore has to be implemented using a loop. The one we will regard as standard is the following, where `temp` is an additional local variable:

```
temp := 0; j:= 0;
while j<= N do
 {temp := max(temp,number(i));
  j := j+1};
 number(i) := temp+1;
```

This seems to the author to corrrespond well to what we would expect of a *maximum* operation, since it accesses each of the things to be maximised once, and only writes once to `number(i)`.

We will give a number of alternatives on page **??** and analyse them.

Quite apart from potential variations in the implementation, are two significant problems when we attempt to verify the Bakery Algorithm using SVA and FDR. The first is a common one: the algorithm is intended to work for any number of threads, whereas a concrete implementation we give can only have a fixed number. The second is that the `number(j)` values range over arbitrary non-negative integers, while SVA can only handle finite subranges of the integers. Indeed it is not hard to see that even with two threads it is possible that the `number(i)`s can grow unboundedly: imagine one always entering the queue waiting for a critical section while the other is performing its.

In this section we will ignore both of these problems (though we will solve them in Section **??**) and study implementations with fixed numbers of threads; and look for errors other than the `number(j)`s exceeding their bounds. Of course you can verify that any chosen bound for the integer type is inadequate by running a check that does look for run-time errors.

The usefulness of the compression strategies described in the previous sections, as well as the problems caused by the state exploration problem, are well illustrated by the following table that gives the number of states found by checks of the Bakery Algorithm for different values of MaxInt and N, the number of threads. The file used is bakeryC.csp, which uses implementation (C) above for the original algorithm and critical sections consisting of signal(css.i);signal(cse.i) as discussed in Section ??.

2.6 Case study: The dining philosophers in SVL

The reader should already be very familar with the dining philosophers problem from earlier references to it in this book (for example ...). All of those were programmed in CSP and relied on synchronous communication to manage the distribution of the resources that are otherwise known as *forks*. We could equally well replace each fork process by a mutual exclusion algorithm, so that each philosopher now has to obtain the right to perform the critical section of both its adjacent ones in order to eat. We can easily create versions of both the symmetric and asymmetric dining philosophers out of the following components each of which uses a mutual exclusion algorithm called Dekker's algorithm [?], see also Exercise ??. As with Peterson's and Hyman's algorithms, there are a pair of threads indexed by 1 and 2. In our new model of the dining philosophers, each fork is replaced by making every pair of adjacent philosophers run a separate mutual exclusion. Thus each philosopher engages in two of these mutexes, and must be in the critical section of both of them in order to eat. The following routines represent what a philosopher has to do to obtain and release one or other end of the jth mutex/fork. The two ends, in common with our representation of Petersons's and Hyman's algorithms, are labelled by the parameter i being 1 or 2.

```
GetCS(j,i) =
b[j,i] := true;
  While b[j,3-i] do
    {if t[j]==3-i then
        {b[j,i] := false;
         While t[j]==3-i do Skip;
         b[j,i] := true;}
  signal(picksup.(j-1+i)%N.j}
```

```
RelCS(j,i) =
   t := 3-i;
   b[j,i] := false
   signal(putsdown.(j-1+i)%N.j)
```

As shown in Figure ??, Phil(j) gets the i=1 side of fork j and the i=2 side of fork (j+1)Nhave added as signals the CSP events that the philosopher would have synchronised with the forks in previous models. The jth philosopher (for i in {0..N-1}) can then be defined

```
Phil(j) = inter{GetCS(j,1);GetCS((j+1)%N,2);
                signal(eats.j);
```

```
                  RelCS((j+1)%N,2);RelCS(j,1)}
```

This is a "right-handed" philosopher because it always seeks the fork (mutex) at index `j` before it does the one at index `(j+1)%N`. A left-handed one would perform these tasks in the opposite order. We know that the parallel composition of these `N` processes can deadlock when implemented in CSP, and must expect some analogous behaviour here. Over CSP this was discovered because the system failed a liveness as opposed to safety property, and it is natural to expect the same here.

We have already seen that safety (i.e. traces) properties are handled in very similar ways in CSP and SVA, both using traces. This is not the case for liveness properties. SVL processes never deadlock in a straightforward sense, since any thread that has not terminated can always perform an action. What actually happens in this SVL version when each has picked up one fork is that they all enter an eternal, but *busy* wait for the availability of the second mutex, going round and round a loop checking for this for ever. The danger is not that the system can reach a state where nothing can happen, but rather that is can reach a state where nothing *useful* can happen.

In this world without deadlock, and where our system's behaviour is unaffected by the external environment, we need to find a different way to specify that systems have liveness properties.

Given that we have labelled everything that represents definite progress with a signal event, it is natural that we might want to prove the following: *from every reachable state, a signal* **will** *occur at some future time.*

However, with the model we have established so far this is not something we can aspire to because there are no *fairness* assumptions in our model. We allow the various threads to interleave completely arbitrarily, including the possibility that one thread will perform an infinite sequence of actions while the rest do nothing. So, for example, if one of a pair of processes involved in a mutex is in its critical section, only the other one may perform actions, and there will take the form of a loop waiting for the conditions allowing it to proceed with its critical section.

We therefore need to weaken our goal to proving that *from every reachable state, there is* **some execution** *that leads to a signal.* This is a reasonable aspiration for SVA processes without fairness.

We have already seen each of these specifications considered in Section **??** when we considered the extent to which LTL could be decided using FDR. For these two specifications $\mathbf{GF}X$ and $\mathbf{G}\neg(\mathbf{G}(\neg X))$, where X is the set of events that we want to be (i) unavoidable and (ii) always reachable at every point in the execution. The first (i.e. the one that our system will not satisfy), proved to be reasonably straightforward (page **??**) while the other (page **??**) needs to be weakened to "every finite trace s of P has an extension of the form $s\hat{}t\hat{}\langle x\rangle$ for some

$x \in X$": *AlwaysPossible*(P, X)

For any nonempty subset X of the signals used in its definition, the symmetric version of the dining philosophers fails *AlwaysPossible*(P, X) because of the classic deadlock in which each philosopher picks up their right hand fork. Similarly the asymmetric version in which there is at least one right-handed and one left-handed philosopher passes each such check.

This is illustrated in the accompanying file `svdphils.svl`. Since this check involves having the whole system on the left-hand side of a refinement check, and therefore normalising it, FDR is able to handle rather smaller numbers of philosophers than it can for more straightforward checks.

The present version of FDR does not handle the sort of fairness that is required to reason about definite progress in shared variable systems of the type dealt with in SVA: what we would need there would be a way to build in an assumption that in any infinite execution each thread performs an infinite number of actions unless it terminates or throws an error. Techniques for handling both this and LTL (as well as other temporal logics) are well understood (see [?], for example) and use automata-theoretic analysis frequently involving *Büchi automata* to reason about fair systems. Until this style of analysis is added to FDR at some later date, and provided it still proves possible to use some effective compressions on at least the thread/variable combinations, we have to conclude that *SVA as a front end to FDR is excellent for reasoning about the safety properties of shared variable systems, but has considerable weaknesses for reasoning about livemess properties.*

This also teaches us that relatively sophisticated techniques for reasoning about LTSs such as those involving infinitary liveness properties and fairness, seem to be much more necessary in the context of systems that, like complete SVL programs, are simply observed by the environment rather than needing to interact fully with it regularly.

The property that all N philosophers do not simultaneously hold a fork is, of course, a safety property and the reader will find that SVA/FDR check this considerably more quickly than the liveness property discussed above.

The author's favourite way of avoiding this situation is the asymmetric ring already discussed in this section. This is not so much because he is left-handed as because it creates a network that does not require an external patch to make it deadlock free, into which category the other well-known resolution of the deadlock falls, namely the *butler* or *footman* who refuses to allow a philosopher to sit down at the table if all his or her companions are already there [?].

We will, nevertheless, study this second solution in the rest of this case study, since it teaches us some interesting lessons about shared variable programming and atomicity. The series of attempts at this model described below appear in separate files `svdphilsb`*i*`.svl` for *i* in `{1,2,3}`.

The natural way to implement this restriction in SVL is to add a counter, accessible by all the philosophers, of how many are sitting down. Before approaching the table, they check that the counter is strictly less than `N-1`, add one to it when sitting down and subtract one when leaving the table. The following is a somewhat naive implementation of this.

```
Phil2(j) = iter{
             if count < M-1 then
               {count := count + 1;
                GetCS(j,1);GetCS((j+1)%N,2);
                signal(eats.j);
                RelCS((j+1)%N,2);RelCS(j,1);
                count := count -1}
                }
```

The reader will find that running this system (with all philosophers initially away from the table and `count` initialised to `0`) creates "out of range" errors for `count`. This is, of course, because the assignments are not implemented atomically and so, for example, the effect of either incrementing or decrementing `count` can be removed by running it in parallel with a second increment or decrement.

The immediate conclusion is that we need to make the statements assigning to `count` atomic, either by using the explicit `atomic` construct or by using a mutual exclusion algorithm such as the Bakery Algorithm. We can, of course, *implement* the `atomic` construct in terms of the others by using a mutual exclusion algorithm that extends the exclusion over *all* the threads rather than just between a subset of them.

Making these assignments atomic, producing a new version `Phil2(j)`, does remove the run-time errors (on the assumption that `count` ranges over `{0..N}`), but does not achieve the objective of preventing all the philosophers from sitting down at once. Therefore the same "deadlock" can still happen. The reason for this is that, when `N-2` philosophers are sitting at the table, the only two non-sitting philosophers can both check `count` at the same time, and find it is `N-2`. They then sit down at the same time and increment `count` twice to `N`.

In order to achieve the desired effect we need to transform our program so that the test of `count < N-1` and the increment that follows if this is successful come within the same atomic section. This seems to require the use of a local variable, as in the following version:

```
Phil3(j) = boolean waiting
```

```
iter{
  waiting := true
  while waiting do
   {atomic{if count < N-1 then
           {waiting := false;
            count := count + 1}}};
  GetCS(j,1);GetCS((j+1)%N,2);
  signal(eats.j);
  RelCS((j+1)%N,2);RelCS(j,1);
  atomic{count := count - 1}}
 }
```

Notice that this now performs one `eats.j` each time round the main loop, whereas the original program sometimes performed none.

This program works satisfactorily. We could add signals into the two `atomic` section to represent the `sits.j` and `getsup.j` events of the CSP model on page **??**.

Exercise 2.6.1 What is the effect on the execution of the system built from `Phil3`s if the whole `while waiting` loop is made `atomic` rather than just its body? Reason about this problem first, trying to spot an undesirable behaviour. Are the signals presently placed in the program sufficient to let SVA show this undesirable behaviour? *Hint: the answer to this question depends on whether you have added the* `sits.j` *event, and if so exactly where you have placed it.* If not, add as many as are needed to do this.

Exercise 2.6.2 Create an SVL program in which the `atomic` sections of `Phil3(j)` are implemented using the Bakery algorithm. How would you expect the times taken to check that all `N` philosophers can never sit at once to compare?

Exercise 2.6.3 An alternative to having `N` processes implement mutual exclusion themselves (as in the previous exercise) is to use one or more extra threads to do it for them.

(i) What should the interface between a single thread process and such an *arbiter* be? Think of each process as having one or more Boolean flags it writes (e.g. "I want to enter my critical section") and the arbiter as having one or more for each of its clients (e.g. "Client `j` can start its critical section").

In designing this interface you might like to consider whether it prevent a single client perfoming two critical sections when the arbiter only meant it to have one.

(ii) Design an arbiter that uses your interface, and which checks its clients, when appropriate, in round-robin fashion to find the next one to allow a critical section.

(iii) Implement this in conjunction with the network of `Phil3(j)`'s, implementing their atomic sections.

EXERCISE 2.6.4 Now design an arbiter that is composed of `N` threads, one coupled with each philosopher. They work by passing a "token" around the ring between them in a single direction, which is implemented by variables that are shared between consecutive threads in the arbiter ring.

How does the behaviour of this ring arbiter compare to your round-robin sequential arbiter?

EXERCISE 2.6.5 Notice that the single process arbiter of Exercise 2.6.3 has something in common with the Butler process we are representing with the variable `count`, in that each of the philosophers has to ask its permission to do something.

Use the techniques, including signalling techniques, you developed for that exercise to remove the `count` variable from philosophers, and place it inside a modified arbiter that now represents the behaviour of the Butler. Naturally the philosophers will now have to ask its permission to sit down and signal it when they get up.

2.7 Notes

The author wrote the first version of the shared-variable compiler in late 2000, and reported this work in [?]. For obvious reasons it was difficult to persuade many others to use a tool whose input language was a CSP type, and so not a great deal happened until David Hopkins's project [?] of 2006/7 in which he built a front end for it. A summary of this work was reported in [?]. The version of SVA presented in this chapter and the next was developed especially for this book, and greatly extends the earlier one, for example by allowing non-constant array index expressions and using compression.

Chapter 3

Understanding shared variable concurrency

In this chapter we look at some more advanced ideas in shared variable programming. First we contemplate a less kind, but arguably more accurate model of shared variables than the rather obvious one used in the previous chapter, and show how the Bakery Algorithm, at least in one of the two commonly seen versions, works nevertheless. We then see a further case study, namely an algorithm (Simpson's four slot algorithm) that attempts to bridge over the difference between the more optimistic and pessimistic versions. We then look at how to derive an appropriate notion of *refinement* between partial SVL programs and use it to study alternative algorithms for computing *maximum* as used in the Bakery Algorithm, as well showing how each SVL program is equivalent to one in which each assignment and expression can be evaluated atomically. Finally we return to the Bakery Algorithm and see how to show that suitable versions of it work for arbitrary numbers of nodes and with unbounded ticket values.

3.1 Dirty variables

Imagine that one process is trying to read a shared variable while another is writing it. What value does it get? Our variable processes `IVAR(x,v)` and `BVAR(x,v)` do not model this situation since reads from and writes to it are each instantaneous CSP events, so do not overlap.

In an unsynchronised implementation without some sort of mutual exclusion to keep reads and writes apart, this is not realistic. We can give SVL an alternative semantics in which either all overlapping reads and writes give a nondeterministic value to the read, or where those of some selection of locations are *dirty* like this. (We will refer to variable of the type modelled previously as *clean*.) To do this we assume that a set of `DirtyVars` has been defined and that for each of these we have events representing the start and end of each write – a read between these two events getting a nondeterministic value chosen from the type of the variable. The easiest way to implement is to have a thread writing to such a variable perform each `ivwrite.i.x.v` or `bvwrite.i.x.v` twice, and to have the corresponding variable process expect writes to come in pairs. That is what SVA does at the time of writing, and what we will assume in the rest of this section.

We assume here that two *writes* to such a variable should not overlap, and make it an error if they do. This is not an issue in the common case where each location is only written by a single thread. See Exercises 3.1.1, 3.1.2 and 3.1.3 for further examination of this issue. The definition of an integer variable therefore becomes

```
IVAR(x,v) = let t = ctype(x)
            Dirty(i) = iveval?_!x?y:t -> Dirty(i)
                     [] ivwrite?j.x?_ -> (if i==j then SKIP
                                           else verror.x -> STOP)
            within
            (iveval?_!x!v -> IVAR(x,v)
         [] ivwrite?jj!x?w ->
              if not(member(w,t)) then verror.x -> STOP
                                  else
                                  if member(x,DirtyVars) then
                                  (Dirty(jj);IVAR(x,w))
                                  else IVAR(x,w))
```

The state `Dirty(i)` is the one in which our variable is giving arbitrary results for reads pending the completion of the write by thread `i` that is currently taking place, and during which it is an error for a different thread to write to this location. This state is ended by the second of the pair of `ivwrite.is` that thread `i` now creates.

Correspondingly, the compilation of a Boolean assignment now duplicates the `bvwrite` event for dirty locations.

```
MainProc(Bassign.(el,e),j) =
                          let Q(lv) =
                          let P(rv) = bvwrite.j.lv.rv ->
                                       if member(lv,DirtyVars)
                                         then bvwrite.j.lv.rv -> SKIP
                                         else SKIP
                              within BExpEval(e,P,j)
                          within BLvEval(el,Q,j)
```

We could, also, of course, spread out reads of shared location into a start and an end, and make the result nondeterministic if any write of the same location happens in this interval. The reason it is not implemented as standard is that the asynchrony already present in the execution model of SVA makes it unnecessary. If a "long" read of this form could overlap a write to the same location (necessarily by another thread), then SVA will find an execution in which the single-event implementation of read occurs during the dirty phase of the write, which of course will return a nondeterministic result.

Long reads are relatively unattractive to implement from the point of view of state space since an `IVAR` or `BVAR` process would need to remember details of each read that is currently in progress.

There is a strong argument for wanting shared variable algorithms in which writes are not protected from other accesses to the same location by mutual exclusion to be correct even when all shared variables are dirty. We could perhaps make an exception for Boolean-valued locations that are only written by a single thread thanks to the following argument (which implies a simple transformation of the process doing the write):

- A Boolean write either leaves the current value the same or it changes 0 to 1 or 1 to 0.
- Since there is only the single process writing the location, it can know in advance of the write whether the value is to be changed or not.
- If it is not going to be changed the write can be cancelled, so there is no dirty phase. Not doing this write is equivalent to a "clean" one from the point of view of all processes reading the location.
- If it is going to be changed it is reasonable (though not completely uncontroversial) to assert that the value read changes monotonically so that, say, one never reads 0 then 1 when overwriting the value 1 by 0. The effect of such a write is therefore the same as a clean one.

We call this strategy *write-when-different* and see an interesting application of it in the next section.

One of the most interesting property of Lamport's original version of the Bakery Algorithm is that it is tolerant of all its shared variables (*number* and *choosing*) being dirty, as the reader can easily check by setting these locations as dirty in the script:

- In input to `svacomp.csp`, dirty variables are introduced as members of `DirtyVars`. For example if set is empty then all variables are clean, and if it is `ivnames` or `bvnames` then it is all integer or Boolean locations respectively.

- In *SVL* .....

The simplified Bakery Algorithm (without *choosing*) satisfies the mutual exclusion specification with clean variables, but does not work with dirty ones. It is easy to see why: a process that is assigning a turn value greater than 1 might create a moment when a second process will read value 0. While this can still happen in the original version, that moment is protected by `choosing[i]` being `1`, which eliminates some of the risk: once `choosing[i]` is set, no other thread attempting to enter a critical section can look at thread `i` and believe that it is not presently seeking or in a critical section until it has left the critical section.

Making *local* variables (i.e. unshared ones) dirty does not make any semantic difference to a program since there is nobody who can read them in the dirty phase. On the other hand, making them dirty certainly increases the number of transitions the variable and thread processes perform as they proceed, and the number states in the state space of the combination of the thread and its variables.

If you look at the state space of running a compressed check with dirty local variables, it will, despite this, be identical to the same check with clean local variables. This helps illustrates why compression works so well in conjunction with SVA. Many of the steps that the combinations of threads and their own variables take are, like this extra step in a local write, things that do not directly affect the external view the combination. They are just computational progress, and `diamond` compression has the effect of eliminating these steps, amongst others, after they have been hidden. The existence of such "progress" steps (such as testing or incrementing loop variables) is very common in shared variable programs, to the extent that just about any non-trivial thread will have many. This helps to explain the effectiveness of compression in SVA as shown in Table **??**.

Making shared variables dirty will usually increase the compressed state space, however, and will *always* give a resulting system that is trace-refined by the one with clean variables.

EXERCISE 3.1.1 Dekker's algorithm, the mutual exclusion algorithm that we used in the the dining philosophers in Section **??**, is described below in the same style we used for Peterson's and Hyman's algorithms in the last chapter. `b[i]` are shared Booleans and `t` is a shared integer variable that ranges over the thread numbers `{1,2}`.

```
iter {
 b[i] := true;
 While b[3-i] do
   {if t==3-i then
```

```
              {b[i] := false;
               While t==3-i do Skip;
               b[i] := true}
  }
 CRITICAL SECTION;
 t := 3-i;
 b[i] := false}
```

Notice that, unlike the other examples we have examined in the context of dirty variables, here there is one (`t`) that is written to by both threads. Analyse it to see if it satisfies the mutual exclusion specification with and without its variables being dirty.

What happens if the two assignments following the critical section are interchanged?

EXERCISE 3.1.2 Analyse Hyman's and Peterson's algorithms to see whether they are guilty of concurrent writes (i.e. ones that might overlap and cause the error event in `Dirty` to occur)[1].

EXERCISE 3.1.3 Suggest some alternatives for the model of dirty variables that bans concurrent writes. You might like to consider the following cases separately:

(i) What if the concurrent writes are all of the same value?[2]

(ii) What if the underlying type is Boolean so that a pair of clashing writes are either the same or cover the entire type?

3.2 Case study: Simpson's four slot algorithm

When one thinks about it, it is a great deal more surprising that the original Bakery Algorithm is tolerant of dirty variables than that the simplified one is not. Dirty variables seem to be a remarkably challenging obstacle to creating a correct shared variable program, at least unless we use mutual exclusion to ensure that no thread ever attempts to read from one while it is being written to, and that no write starts while it is being read from. The device used in the original Bakery Algorithm of having a `choosing` flag that protects a write can be seen as a weak version of

[1]We have already seen one mutual exclusion algorithm, namely the Bakery Algorithm, in which no location is written by more than one thread. A second is Knuth's algorithm [?], an implementation of which can be found with the examples accompanying this chapter.

[2]In the theory of concurrent algorithms, researchers sometimes consider a parallel machine model in which multiple processors are allowed to make simultaneous writes to the same location, but only if all the written values are the same: a Parallel Random Access Machine with *Common* Concurrent Writes, see [?].

this: it prevents a second thread reading the protected variable while it is being written to provided the read starts after the write. It can be seen that either this or full mutual exclusion can easily have the effect of holding up a thread that wants to read a location. In one case it has to wait until it can perform its critical section of accessing the location; in the other it has to wait until `choosing(j)` is unset before it can rely on the corresponding `number(j)`. Waits like these have the potential to create "deadlocks" in which each thread is prevented from getting any of mutual exclusions it wants for reading or writing in a way similar to what we saw in Section **??**, the shared-variable implementation of the Dining Philosophers.

Simpson's four-slot algorithm [**?**] is an attempt to rectify this: it assumes there are two processes, one of which can write to a shared variable at any time and one of which can read from it. It seeks to create a clean shared variable of a given type from four dirty ones of the same type and several Boolean flags. Separate processes may write and read the variable at any time, and the algorithm seeks to ensure that slots read from and written to simultaneously are different, but that where reads and writes do not overlap it behaves like a variable process with atomic reads and writes.

In the following model the `Reader` and `Writer` processes just perform these tasks over and over. In a real application the body of each loop would be called when an enclosing thread wanted to read or write the variable, but of course we should bear in mind that the threads below are interleaved completely arbitrarily and so can simulate any such real application.

```
Writer =
{iter
   {wval := inp;
    wpair := !reading;
    windex := !index[wpair];
    slot[wpair,windex] := wval;
    index[wpair] := windex;
    latest := wpair;
   }
}

Reader =
{iter
   {rpair := latest;
    reading := rpair;
    rindex := index[rpair];
    outp := slot[rpair,rindex];
   }
}
```

Here, `inp` represents the value that want to be written (and which we assume here is a value set by another thread) and `outp` is the value returned by the read. The array of `slots` are shared between these two threads, as are the four Booleans `reading`, `latest`, `index[true]` and `index[false]`. Apart from `reading`, which is written by `Reader`, they are all written only by `Writer`. The rest of the variables are all local to one or other thread.

The author finds it more difficult to get a clear intuition for this algorithm than, for example, the Bakery Algorithm. We can, nevertheless, see that it should satisfy a number of useful properties such as the following:

- If a *read* is executed in the interval between two iterations of the `Writer` loop, then the most recently written value is read.
- If a *read* pauses then the value written is, thanks to `wpair := !reading;`, to a different slot to the one that is being read from.

It is not easy to understand the dance that the two threads perform in general, though, particularly if we assume that the four Boolean flags are dirty.

Fortunately it is easy to adapt the above program not only to run on SVA but also to analyse it. The only issues about adapting it is that, above, `slot` is a *two*-dimensional array indexed by Booleans, and `index` is also indexed by Booleans. We can convert this to the SVL language by firstly interpreting all the Booleans in the above programs as integers `0=false` and `1=true` and secondly replacing `slot[a][b]` by `islot[2*a+b]`. Negation of a Boolean becomes `1-x` for the corresponding integer.

Consider first the claim that there is never a read and a write performed on the same slot simultaneously. This is easily verified by making all of the shared locations dirty, setting the type of data (i.e. what is stored in the `slots`) to be `{0,1}`, initialising `inp` to be `0` and never letting it change, and checking that `outp` never takes the value `1`. Clearly this would be possible just when `Reader` reads from a slot that it is in its dirty phase. The fact that this specification (i.e. that `1` is never read) holds seems, as with the Bakery Algorithm's tolerance for dirty variables, rather remarkable.

Since our program is obviously strongly data independent in the type of data (stored in `slots`, `inp`, `wval` and `outp`), this evidently proves this *no collision* property for all values of this type.

A second property we would like to hold is that every read returns a value that, while it may not be the *most* recently written value, is one that is either *being* written or at some *recent* time was the most recently written one.

It seems reasonable to interpret the italicised *recent* in the last sentence as meaning "during the execution of the current read cycle". Therefore what we actually demand is that the read value may be any one of

- The value of the most recently completed write at the start of the current read cycle.
- The value of any write that was happening at the start of the current read cycle.
- The value of any write subsequent to the above.

Again relying on data independence, we can easily check whether this holds by arbitrarily writing `0` or `1` on each write cycle and modifying the `Reader` and `Writer` as follows:

- There is a new integer variable `lastw` that remembers the value of the last completed write.
- There is a new Boolean array `okread` indexed by the data values `{0,1}`: at the end of each read cycle `okread(i)` tells us whether a read of value `i` is legitimate under the above specification. At the start of each read cycle `okread(i)` is set to `true` if either `wval` or `lastw` takes value `i`; otherwise it is set to `false`.
- Each time a write cycle is performed, `okread(wval)` is set to `true`.

Regrettably, this specification fails with all the locations dirty, though it succeeds if the flag variable `latest` is removed from `DirtyVars`. (See the file `Simpson2.svl`.) Furthermore the trace that SVA discovers when `latest` is dirty involves this variable having the same value being written to it that it already had before the write: the error involves the opposite value being read during the dirty phase of the write.

This suggests that, for this and perhaps the three other shared Boolean variables, we might employ the write-when-different trick described on page **??** and only assign to them when they are are to change. This modification makes the above *recentness* specification true even if we still interpret the same variables as dirty. (Recall that we argued there that there was a reasonable case for asserting that a Boolean variable treated in this way might be relatively easy to implement as clean: we are not relying on this here.)

There is one further specification that it is natural to demand of any reasonably model of a shared variable: it must be *sequentially consistent*, meaning that if we read a series of values from the variable and remove consecutive duplicates then we get a subsequence of the initial value of the variable followed by the values written to it in order. In other words we cannot carry out two consecutive reads and get an older value on the second read.

The strong data independence of the algorithm in the type of data means that we can run a rather cunning check for this. We arrange that the initial value of the variable is `0` (every `slot[i]` equals this) and that the sequence of writes performed is a completely arbitrary number of `0`s followed by an arbitrary number of `1`s. If it were possible to get an out-of-sequence read value then it would be possible to read a `0` after a `1`, because the there is an execution in which the newer value that gets read co-incides with the very first `1` that gets written. In other words, if the *nth* member of the input sequence can be read before the *mth*, where $n > m$, we consider the case where the first `1` to be input appears at place n.

We can therefore arrange a check on this basis: the input value `inp` is initialised to `0` and assigned (cleanly[3]) `1` at an arbitrary point in the execution by a third thread, and we check if `outp` is ever `0` when it has previously been `1`.

This check succeeds if all the Boolean flags are clean, but fails if any one if them is write-when-different but dirty (meaning, say, that on writing one of them that is `0` to `1`, consecutive reads might see `1`,`0` before it settles to `1`). To run these checks, see `Simpson3.svl`.

At least for the three flags set by `Writer`, there is an interesting explanation for this problem. It is easy to see that each one of these flags can directly affect the choice by `Reader` of which slot is read from next. Suppose that the `Writer`'s execution gets suspended mid-way through such a critical write, which when complete will tell the `Reader` to look at a more newly-written `slot`. Now suppose that the `Reader` runs through twice in this period, the first time getting the changed flag value and the second time getting the old one. Since the `Reader` uses no state that it has set prior to a particular read, it will inevitably get the value from the new `slot` on the first read and the old one on the second. This argument demonstrates that both Simpson's algorithm and any other with a similarly "memory-less" `Reader` cannot have the sequential consistency property in the presence of the dirty flag variables, when the dirty period can extend over two read cycles like this.

The author knows of no way of solving this problem by the addition of further slots or Boolean flags. We can, however, solve it by building on what we have already established about Simpson's algorithm when `latest` is made write-when-different, namely that access by the two processes to the slot avoids all dirty reads and that the value read by the reader is always recent. The essential idea behind our solution

[3]It is legitimate to make clean writes to locations such as this and `okread(i)` that are shared between threads and introduced by us for specification purposes, because they are not part of the algorithm itself, merely tools to help us observe it.

is to augment each slot with an integer counter that is always written or read along with the "real" slot value. We know (i) that these counters and the slots read cleanly and (ii) that each counter/data value combination is recent. If each write cycle then increments and writes the counter: the first write getting counter `1`, the second `2` and so on, it should be easy to detect out-of-sequence reads.

We can therefore implement the following modified algorithm:

```
Writer2 =
{iter
   {wval := inp;
    counter := counter+1;
    wpair := !reading;
    windex := !index[wpair];
    slotcounter[wpair,windex] := counter;
    slot[wpair,windex] := wval;
    index[wpair] := windex;
    if latest!=wpair then latest := wpair;
   }
}

Reader2 =
{iter
   {rpair := latest;
    reading := rpair;
    rindex := index[rpair];
    thiscounter := slotcounter[rpair,rindex];
    if thiscounter > lastcounter then
                 { outp := slot[rpair,rindex];
                   lastcounter := thiscounter};
           }
   }
```

Note that this `Reader2` does not read a `slot` that was written no later than its previous one Of course the `slotcounter` locations should be treated as dirty, but we know this will have no effect thanks to the established properties of Simpson's algorithm. Given this, it is obvious that the revised algorithm never returns an out-of-sequence result. It is also clear that the value of any read is always at least as up to date as the one produced by the original algorithm.

It is therefore fair to argue that we have *already* verified the cleanness and recentness properties of this algorithm, and that the no-out-of-sequence property is obvious. This is fortunate since the revised algorithm is, thanks to the counters, infinite state.

We can nevertheless the revised algorithm directly for a small number of writes: whatever value we pick for `MaxI`. Later in this chapter we will see techniques that can prove both the similarly limited Bakery Algorithm and this one in general.

In a practical implementation, in order to work in the presence of completely arbitrary interleaving, the counters need to be large enough to hold the value of the maximum number of writes that could conceivably occur during a run: presumably 64 bits would be sufficient in most cases. With care it should be possible to use smaller values when the length of time that a variable can remain dirty is bounded. Of course if we can build Boolean flag variables in which a write-when-different is effectively clean, there is no need for the elaboration with counters.

EXERCISE 3.2.1 Construct a version of Simpson's algorithm that tests the following specification: *the value of each read is one of the four most recent writes.* (Make an appropriate adjustment at the start of the run when there have been three or less writes.)

Even with all locations clean, it should not satisfy this specification. How can we be sure that any algorithm that never blocks reads or writes, which is arbitrarily interleaved, and which prevents dirty reads of values of the data type, can never satisfy this specification even if "four" is increased?

EXERCISE 3.2.2 Notice that Simpson's algorithm is limited to a single reader and a single writer of our variable. Discuss the issues that would be involved in increasing these numbers, and see if you can solve the problem. Could a number of reading and/or writing processes share the use of a single `Reader/Writer` agent?

It could be argued that in a fully asynchronous world this latter technique is less good that having the corresponding code embedded within the thread that actually creates or consumes the value, because the reading and writing agents may not get scheduled sufficiently often. Do you agree?

3.3 Observing and refining SVL programs

One of the most interesting properties of any translation of another language into CSP is that it automatically gives a definition of *refinement* between programs: refinement between terms in a language like CSP holds just when it does between the CSP translations.

As explained in Section **??**, the only form of refinement check that usually makes sense for SVA is over finite traces, since the CSP processes it creates can frequently diverge thanks to unfair executions. Aside from the use of `AlwaysPossible` on page **??**, every check we have used for SVL has been of the complete system against a fixed CSP specification constructed in terms of `Errors` and signals to judge its correctness.

There is, of course, a second important role that we would like refinement to have, namely helping us to judge when one *partial* program refines another: can we replace the *maximum* algorithm we selected for the Bakery algorithm by another without introducing any undesirable behaviour?

The compiler as described to date treats the program supplied to it as a closed system. In particular, its locations are neither read nor written to by any entity outside this closed system. To judge a partial program, one that will fit into a program context like the one where the maximum algorithm is placed, we need to take into account how it interacts with the rest of the program through the shared use of locations.

The refinement relations described in this section are implemented in a special file, `refsva.csp`*, which acts as a front end to* `svacomp.csp`.

There are actually three sorts of context we need to consider:

- In $C[P]$, P only appears within `atomic` statements[4] created by $C[\cdot]$. *In such contexts, parallel processes cannot read or write P's locations while it is running. So P is only influenced by the initial values in its locations and only influences other parts of the system via the values it leaves behind in locations it can write and they can read. Semantically speaking, such P can be viewed as functions from their initial state to the combination of their final state and the possibly empty sequence of signals produced.* We will call this a *sequential* context.
  When reasoning about such a P, we need to tell SVA which of its locations might be written to by preceding code or read by subsequent code. For this purpose the user defines two sets of locations: `SeqWrites`, which contains all locations where writes by preceding code might leave a value other than the initialisation to be read by `P`, and `SeqReads`, which contains all locations that `P` might write to and subsequent code read from. It is important that these sets contain not only those locations that might be read or written by the rest of the thread within which `P` sits, but also the relevant ones in other threads, made sequential by the `atomic` statement surrounding `P`.
- In $C[P]$, P only appears at the level of the final parallel combination: in other words it plays the role of one or more complete threads. *In such contexts, the initial values of locations are all defined statically by the context or by P itself, and there is no process that runs after to use the final values. However some other threads may be able to read and write to locations used by* `P`. *We therefore model P in much the same way as for complete programs, only*

[4]This type of context also applies to any case where there is no parallelism in the complete system.

allowing the environment (playing the role of these other threads) to read the locations ParReads *and write to* ParWrites[5]*.* We will call this a *parallel* context.

- If P appears as a part of a single thread but is not forced to be atomic by that thread, then we need to consider both of the sorts of influences discussed above. This is a *general* context. There is no need in this case to include locations written by other threads in SeqWrites or ones read by other threads in SeqReads, since the interfaces required by ParReads and ParWrites will give the environment all relevant access to these.
- For the last two of there, there is a further parameter that defines how our term interacts with other threads: ext_atomic is a Boolean that is true when we want to allow external threads to go into atomic sections, preventing any activity by our term during that period. We can see that this is necessary from the following example: assuming that x and y are initialised to 0, and that ParReads={x,y}, ParWrites={} and SeqWrites={}, consider the following pair of programs:

```
PP: iter {x := 1;      QQ: iter {x := 1;
          x := 0;                y := 1;
          y := 1;                x := 0;
          y := 0}                y := 0}
```

  If ext_atomic if false, then these two programs refine each other in a parallel or general context. Any read of x or y can get the values 0 or 1. If, on the other hand, ext_atomic is true, then PP is a proper refinement of QQ, which can give both variables the value 1 during the same atomic section, something impossible for PP. This example can be found in extatomic.svl.

For both sequential and parallel contexts, we need allow the environment to assign and read some of P's locations. In the sequential case, assignments are only possible before P runs and reads are only possible after P has finished, whereas in parallel contexts there is not restriction.

What we therefore do is implement a channel along which an "external thread" can write to locations in a set ExtWrites and read from those in ExtReads. This will give us exactly what we need to consider parallel contexts with ExtWrites=ParWrites and ParReads=ExtReads, and be adaptable to handle sequential contexts and general ones. SVA allows such reads and writes as though they were performed by the

[5]In the case of integer locations we only consider the case of writes of values within the content type of the given location, since, thanks to our error regime, no thread will attempt any other write.

“thread” with index -1. It prevents any such reads and writes when the process P we are considering is within an atomic section, just as it would the threads that the environment is playing the part of. For this reason, provided that P obeys the rule that no atomic section sends more than one signal, there is no need to make visible the start_at.j and end_at.j events that code inside P performs.

When ext_atomic is true, the external thread can enter and leave atomic section via start_at.-1 and end_at.-1.

For example, if we were comparing two implementations of thread i in the original Bakery Algorithm, it would be natural to define

```
ParReads = {number(j),choosing(j) | j <- {1..N}}
ParWrites = {number(j),choosing(j) | j <- {1..N}, j!=i}
ext_atomic = false
```

Recall that when we introduced the implementation of *maximum* on page 60 we said this was one of many alternatives. The following are four alternatives, with the previous one being Maximum A

```
Maximum A:
temp := 0; j:= 0;
while j<= N do
 {temp := max(temp,number(i));
  j := j+1};
 number(i) := temp+1;
```

```
Maximum B:
j := 0;
while j<=N do
 {number(i) := max(number(j),number(j))
  j := j+1};
number(i) := number(i)+1;
```

```
Maximum C:
j:= 0;
while j<= N do
 {temp := number(j);
  if temp > number(i)
     then number(i) := temp;
  j := j+1};
 number(i) := number(i)+1;
```

```
Maximum D:
j := 0;
while j<=N do
  {if number(j) > number(i)
      then number(i) := number(j);
   j := j+1};
number(i) := number(i)+1;
```

On the assumption that the number(i) only take non-negative values, these are all equivalent as programs in a sequential language, which is the same as saying that they are equivalent in SVL if each of them is made atomic, or that they are equivalent in sequential contexts. Indeed we can verify this by setting SeqWrites to {number(j) | j <- {1..N}}, SeqReads to {number(i)} (ParReads and ParWrites not being relevant to sequential contexts). Note that the number(j) (j!=i) are included in `SeqReads` because the other threads may have assigned to these before the atomic section begins.

There are subtle differences between the algorithms, however, when parallel threads are allowed to see and alter the values of the `number(j)` locations, as is the case in the Bakery Algorithm. We will assume that the `temp` and `j` variables are local to the particular thread and cannot be seen outside it.

We have two choices of how to compare these algorithms. The first is to compare versions of the complete thread `i` with the various `Maximum` algorithms substituted, and the second is compare the algorithms themselves in the way appropriate for a general context. Example files are provided which do this both ways, for particular values of `i`, `N` and `MaxI` that can be adjusted by the reader.

In trying these out in a prototype framework the author realised there was a problem. Run-time errors are quite common in these systems because of the way 1 is added to a value that can easily be the largest possible for the type of the `number(j)` and then assigned back to a location of that type. For one system `P` to refine another one `Q`, it is entirely reasonable to demand that when `P` generates an error event then so can `Q`. The *problem* is that the counterexamples found sometimes arise from events that *follow* error events: in other words `P` and `Q` both perform a trace of the form `s^<error.i>`, but `Q` can then perform some other event that `P` cannot. It seems unreasonable to differentiate processes on the basis of what happens *after* an error. It seems clear that we should not distinguish between processes if their only differences follow run-time errors, and so the framework for refinement that SVA uses prevents all events that follow members of `Errors`. It is reasonable to term this assumption *weak error strictness*, by analogy with divergence strictness. We will introduce the strong form in Exercise 3.3.2.

This and other issues related to deciding questions of refinement between SVL terms in any of the three sorts of context are addressed by

`refsva.csp`, the refinement front end for `svacomp.csp`. It is this file that uses the parameters `ParReads`, `SeqWrites` etc.

When building the model of a `Cmd P` for a general context, we need to know, when looking at a counter-example trace, what the initial state of `P` was. This is because when we compare the traces of `P` and `Q`, we need know if a given behaviour of `Q` started in the same state or not. We should therefore allow writes to the locations `SeqWrites` at any time strictly *before* starts running, with any location that is not written to in this way taking the value that the program initialises it to. These have to be left visible in the trace so that when we are testing the refinement of one program by another, we always compare them from identical starting states. To mark the end of this initialisation phase we introduce a special signal `PStart` that the model communicates at the point that `P` starts running. Similarly, to distinguish the final state of `P` in which `SeqReads` can occur, we introduce a signal

PEnd that it communicates when it has terminated. The SVL program that is run is therefore signal(Pstarts);P;signal(Pends). We can then put this running code in parallel with a CSP process:

```
GRefReg0 = PStart -> GRefReg1(false)
          [] ivwrite.-1?x:inter(SeqWrites,ivnames)?_ -> GRefReg0
          [] bvwrite.-1?x:inter(SeqWrites,bvnames)?_ -> GRefReg0

GRefReg1(ineat) = PEnd -> GRefReg2
          [] ivwrite.-1?x:inter(OTWrites,ivnames)?_ -> GRefReg1(ineat)
          [] bvwrite.-1?x:inter(OTWrites,bvnames)?_ -> GRefReg1(ineat)
          [] iveval.-1?x:inter(OTReads,ivnames)?_ -> GRefReg1(ineat)
          [] bveval.-1?x:inter(OTReads,bvnames)?_ -> GRefReg1(ineat)
          [] ext_atomic and (not ineat)&start_at.-1 -> GRefReg1(true)
          [] ext_atomic and  ineat&end_at.-1 -> GRefReg1(false)
          [] (not ineat)&([] x:Union({Signals,ISignals}) @ x -> GRefReg1(ineat))
          [] (not ineat)&([] x:Errors @ x -> STOP)

GRefReg2 =  iveval.-1?x:inter(SeqReads,ivnames)?_ -> GRefReg2
          [] bveval.-1?x:inter(SeqReads,bvnames)?_ -> GRefReg2
```

This notices the initialisation, running and final state phases and allows the appropriate external accesses to the locations, and also stops P when an error occurs. The parameter on the middle state remembers whether the external state has claimed an atomic section. If it has, then the activities of the process we are modelling are prevented.

We are now in a position to test the refinement relation between the different Maximum algorithms. This is implemented in the file CompareMax.svl.

The model of a sequential context is equivalent to this with ParReads and ParWrites both being {}. For parallel contexts we do not need the PStart and PEnd signals. It is this that we can use to check the effects of how a complete thread of the Bakery Algorithm is affected by the different Maximum algorithms: we can compare them for refinement.

So what of our Maximum algorithms? SVA and FDR can only test refinement between them for fixed values of i and N, and fixed finite type of integers stored in the number(j)s. If one tests them for N = 3 and then for i=1,2,3 separately, with MaxI=4, one find that:

- All four are indeed equivalent when the algorithms are which forces the algorithm itself to be executed atomically.

- When not atomic, `A` properly refines `B, C, D`; `Maximum D` is properly refined by `A, B, C`. `B` and `C` are equivalent.
- When checked in the context of a thread of the original Bakery Algorithm...
- For the simplified algorithm...

The reader will find that refinement comparisons of terms take significantly longer than checking some simple property of one or other system. Just as with the `AlwaysPossible` check in Section **??**, this is because a more complex process has to be normalised.

Quite aside from the three basic types of refinement that we have defined, depending on the sort of context, we have also shown how to tune the refinement question according to the assumed abilities of the rest of the program to read and write shared locations and go into atomic sections. There is fortunately a simple result that links all of these ideas.

Theorem 3.3.1 *When* `P` *and* `Q` *are members of* `Cmd`*, then*

(i) `P` *sequentially refines* `Q` *for given sets* `SeqReads` *and* `SeqWrites` *if and only if the same general refinements apply with both* `ParWrites` *and* `ParReads` *empty, and* `ext_atomic=false`*.*

(ii) `P;Iter Skip` *parallel refines* `Q;Iter Skip` *for given sets* `ParReads` *and* `ParWrites` *if and only if the same general refinements apply with both* `SeqWrites` *and* `SeqReads` *empty, and* `ext_atomic` *the same in both cases.*

(iii) Suppose the general refinement holds for a quintuple `(SeqReads,SeqWrites,ParReads,ParWrites,ext` *and a second quintuple* `(SeqReads',...,ext`*atomic')+ is* contained *in the first, in the sense that* `SeqReads'`$\subseteq$`SeqReads` *etc and* `ext_atomic' => ext_atomic`*. Then the same refinement holds over the second quintuple.*

(iv) Exactly analogous results hold for sequential and parallel refinement, with pairs and triples of parameters respectively.

The first two parts of the above result follow directly from the structures of the contexts used to test the refinement. In part (i) they give identical behaviours. In part (ii) they give identical behaviours except that the general one inserts `PStart` at the beginning of each trace. The reason for the sequential composition with a non-terminating process in this part is because parallel refinement ignores termination, whereas general refinement does not, since it is detectable from the event `PEnd`.

Part (iii) and (iv) are are straightforward once we realise that the CSP model we get of `P` with the weaker set of observations (i.e. `SeqReads'` etc) can be obtained from that with the stronger one by putting it in parallel with *STOP*, synchronising on all the events (reads, writes, `start_at.-1` and `end_at.-1`) that the observer is allowed to try in the stronger environment but not in the weaker.

EXERCISE 3.3.1 What refinement relations hold between the following simple programs in a general context, where `SeqReads`, `SeqWrites` and `ParReads` are all `{x}`, and `ParWrites = {}`. Assume `MinI=0`, `MaxI = 5`, and `x` is of type `MinI..MaxI}`.

A. `x := 1`

B. `x := x; x := 1`

C. `x := 1; x := 1`

D. `x := 1; x := x`

E. `x := x/x`

F. `x := (x-x)+1`

F. `x := (x+1)-x`

Would your answer change if `ParReads` became `{x}`?

EXERCISE 3.3.2 Recall that we made programs *weakly* error strict on page **??** by insisting that no event follows an error. The best analogue in standard CSP for this is the fact that no event every follows ✓. A better analogue for the divergence strictness of some CSP models would be to insist that as soon as a process becomes able to communicate an error signal it is immediately equivalent to `CHAOS(Events)`, the process that can perform every legitimate trace.

Define and implement (in terms of existing operators) a CSP operator $\mathit{StrictTr}(A, P)$ such that whenever the process P performs any member of A it can also perform any trace at all. Thus, for example, $\mathit{StrictTr}(a, b \to a \to Q)$ should be equivalent to $b \to \mathit{CHAOS}$ over the traces model $\mathcal{T}$. [You might find the *throw* operator Θ_A useful here, but it is no essential over $\mathcal{T}$.]

If we routinely made our refinement models of SVL terms strongly error strict by applying `StrictTR(Errors,.)` to them, how (if at all) would your anawer to Exercise 3.3.1 change?

3.3.1 The Bakery Algorithm revisited

If we replace one term with one that refines it inside some context, then the context will continue to refine any specification it satisfied before. If, on the other hand, we replace it by any other term, we cannot be so sure.

Since `Maximum A` is incomparable under the refinement relation with the others, we simply do not know at this stage whether either the simplified or original Bakery Algorithm will continue to satisfy mutual exclusion when the maximum is replaced by any of the others. In the author's experiments involving up to 5 threads with with `MaxI` up to 12, all of the versions did satisfy mutual exclusion with the exception of the original algorithm with `Maximum D`. However *both* the simplified and original algorithms fail the specification with these parameters when the line `number(i) < number(j)` of `Maximum D` is replaced[6] with `number(j) >= number(i)`: an apparently innocent change that would not change the outcome of the conditional statement in a sequential program.

This actually conceals two distinct alterations in the behaviour of this thread as in the context of shared variables: it reads the values of `number(i)` and `number(j)` in the opposite order, and in the event of equality carries out the assignment.

The latter of these in particular is anything but innocent since we find that after the comparison has been carried out with equal values, thread `j` might have altered, even to `0`, the value that gets assigned to `number(i)`. We find that it is therefore possible to run the changed algorithm in the context where, say, `i=2` and `number(1)` stays at value `3` throughout, but thanks to `number(3)` changing from `3` to `0` at a crucial moment the value of `number(2)` is assigned `0` before the program look at `number(4)`.

But of course exactly the same could have happened had `number(3)` changed from `4` to `0`, and the original `Maximum D` would also have committed the error. On reflection, version `D` should not be regarded as a reasonable maximum algorithm in a shared variable environment, since the type of behaviour discussed here means that it might return a value as the supposed maximum which is less than one of the `number(j)` that never changes while it runs: there is no way that we can therefore claim that the failure of the Bakery Algorithm, when it is used, represents a problem with the original descriptions of the algorithm on pages **??**s and **??**.

There are, of course, two further versions of the comparison above that the reader can try, namely ones that just make one of the two changes discussed above: the results are different for these, too.

What all of this does show is how remarkably difficult it sometimes is to understand and reason about shared variable programs. One reason for this is that one thread can be influenced by what others do at just about any time. This is different from CSP, where the only influences on a process are the events that it explicitly performs.

It also shows how much we need methods that are capable of proving versions of the Bakery Algorithm for all numbers of threads and arbitrary integers. See Section **??** for these.

[6]Furthermore, this replacement only created a problem with at least 5 threads.

3.3.2 Atomic equivalent programs

Programs would be easier to understand if we did not need to break down individual program statements (assignments etc) into multiple events. What we will show in this section is that every SVL program can be transformed into one which is equivalent and where the only uses of shared variables occur in `atomic` assignments. The actions of any such program are therefore determined by how the assignments and signals performed by different threads interleave.

It is intuitively clear that where an assignment `le := re` can use no shared location it is equivalent to `atomic {le := re}` because whether or not it is punctuated by reads and writes by other threads does not matter. The location represented by `le` is not changed by any of the external writes, and neither is the calculation of the expression `re`. Finally, it does not matter when an external read of any of this thread's shared locations happens during the period when `le := re` is being executed, since it does not change any of them!

Now suppose that in carrying out our assignment we execute no more than a single evaluation or write of a shared location during its execution, and that if it is a write then the location is clean. That was stated carefully, because in fact, thanks to array indices (which may of course be nested), the separate evaluation of the expressions on left and right of `:=`, and the fact that we may be writing to a location we have already read from, there is no bound on how many times a single assignment can access a given location. Imagine an execution in which our assignment was carried out atomically at a particular point in the overall sequence of reads and writes. Then any non-atomic execution in which the single shared access happens at the same time is equivalent in the sense that the same value is assigned to the same location and the state visible to other threads is the same.

We can thus claim that any such assignment is not changed in its effect by making it atomic. This claim can be generalised. Suppose a thread reads a location `x`, that other threads can read from but not write to, as part of executing an assignment. Then the effect is no different whether `x` is read as part of the assignment itself or if it reads local `lx`, where `lx := x` has been performed immediately before: the value read is certain to be the same in either case.

We therefore deduce that any assignment in which at most one location that can either be written by another thread, or is written by this assignment, appears is equivalent to running it atomically.

We cannot prove this in general on SVA but we can prove the equivalence for any particular example, using refinement.

Note that if some components of a given array are shared and some are not (unusual but possible), it may not be possible to determine in advance whether a particular assignment actually accesses a shared location. Similarly it is not in general possible to determine, in advance of executing an assignment, which location(s) will be accessed.

If our program had assignments only of this limited shared access form, and where no shared location appears in any other place in the program (and they could occur in the Booleans controlling `while` and conditional constructs, and in `isignal` expressions), we would have achieved our overall goal, modulo the cleanliness constraint. We will show later how to overcome this, but for the time being we will assume that all locations are clean in our examples.

It is actually reasonably straightforward to transform any program construct so that it conforms to this pattern, and indeed to further require that the Booleans controlling `while` and conditionals, and the integers output by `isignals` are single local variable. We will call this, when all the assignments are indeed made `atomic`, the original program's *atomic equivalent form.*

We will first consider how to transform a general assignment. What we do is create a program that makes exactly the same fetches and performs exactly the same evaluations in the same overall order, and then writes the same value to the same location, but which obeys the rule that each line follows the above rules. `svacomp.csp` first evaluates the location to be assigned to, then the value to be assigned, and then performs the write, so that is what the transformed code must do too. In order to work out which fetches it performs and in which order, one has to study the expression evaluation code, and especially that for working out which `fetch` to perform and substitution. In the transformed version we make all such fetches to new local variables, and when an expression has to be evaluated (either to calculate an array index or the final assigned value) to do that in a separate line, assigning the result to a further local variable. Thus the assignment `a[x+a[x+1]] := y+x*y`, where all the locations mentioned are fully shared, could be transformed into

```
lv1 := x;          -- the leftmost x but also gets
                   -- substituted into a[x+1]
lv2 := lv1+1       -- inner array index
lv3 := a[lv2]      -- value of a[x+1]
lv4 := lv1 + lv3;  -- the index in a[] to which assignment
                   -- will be made, completing LHS
lv5 := y;          -- fetched because of leftmost,
                   -- but substitutes for both y's
lv6 := x;          -- x on RHS
lv7 := lv5 + lv6*lv5   -- value to be assigned
a[l4] := lv7       -- the actual assignment
```

where the `lv`*i* are local variables. This program is equivalent because it performs the same fetches in the same order and then writes the same value to the same location. You can prove the equivalence (without any restrictions on dirty variables) of the original assignment and this program using general refinement, and find a similar expansion for any other assignment. Since we know that each line of the above program is equivalent (under the restrictions) to its own atomic version, it follows that the original assignment is similarly equivalent to the above program with `atomic` added separately to each line.

If `x` were only written by this thread, there would be no need for the separate assignments to `lv1` and `lv6`, with `x` being used in place of these local variables.

The crucial part of this transformation is that no line that we add `atomic` to generates more than one effect that can pass information in or out of the system namely an external write or read, a signal or an error event. This is because, if two were generated, they could be split in a non-atomic execution but not in an atomic one, meaning that the two would in most cases not be equivalent. If we know that no run-time error will occur, there is no good reason to evaluate expressions such as assigned to `lv4` and `lv7` on separate lines.

There is nothing magic in the particular way in which SVA evaluates assignments that enables them to be transformed in this way. For example if SVA were changed so that it evaluates the right-hand-side of an assignment first, the above transformation could easily be changed to reflect this. See Exercise **??** for insight into this.

It is easy to remove all non-local variables from an `isignal` or conditional: `isignal(a,e)` just becomes `lv := e; isignal(a,lv)` and `if b then P else Q` becomes `lbv := b; if lbv then P else Q` for new local variables `lv` and `lbv`. It is easy to see that in each of these pairs of programs, aside from accesses to local variables, the two processes perform exactly the same actions. Similarly, `while b do P` can be replaced by `lbv := b;while lbv do {P;lbv := b}`.

If such a transformation is carried out on every one of these three constructs that uses a shared location, we can obtain an equivalent one where the only uses of shared variables are in assignments, and all conditionals, loops and signals use only a single local variable. If we then use the above strategy to convert every assignment possibly using more that one shared variable to a sequence, we have a program where every assignment and signal can be made atomic, and where all conditional decisions are in fact made atomically.

There is in fact little problem in extending this model to allow for dirty variables. We first note that the above transformation of an assignment into multiple steps is valid both for clean and dirty locations. The problem comes when we make the final line, which does the assignment itself, atomic, since other threads are then banned from reading the location during its dirty phase. The solution comes in two parts: in order to achieve the desired effect when other threads *read* this location, we replace `sdl := lv` (the last assignment of the result of the transformation above, necessarily to a shared dirty location from a local variable) by

```
atomic {sdl := randt};
atomic {sdl := lv}
```

where `randt` is a dirty variable with the same type as `lv` assigned by the thread `iter {randt := randt}`, so any read from it gives a random value.[7]

If only one thread can write to `lv` (so no write conflicts can arise), the above is sufficient. Otherwise we need to "shadow" the location `sdl` with a (local) boolean flag `bsdl` (initially false) and replace the above by

```
atomic {if bsdl then Error
             else {bsdl :=  true;
                   sdl := randt}};
atomic {sdl := rv;
        bsdl := false}
```

where `Error` creates an error message to simulate the one that our model of a write conflict to a dirty variable creates (page **??**).

Note that since the location `sdl` is completely determined by the time the final assignment of the original transformation takes place, we can (i) always tell if it *is* dirty and (ii) can be sure that the location written to will not change between the two atomic segments implementing the dirty assignment.

It is obvious that none of our transformations can affect whether an error action can take place in a system. With the possible exception of the very last transformation above (reproducing the error created by a write conflict), the actual error event reported is exactly the same.

The fact that the results of our transformation *are* equivalent to the original program is illustrated in several accompanying files. Of course in defining "equivalent" here, no local variables may be visible from the outside; but there are no other restrictions. We will see an interesting application of these results in Section **??**.

One of the benefits of the style in which we have cast refinement and therefore the equivalence of SVL terms is that programs using `atomic` sections may be equivalent to ones without. This is because we have not had to make the beginnings and end of internal atomic sections visible to the outside world.

Indeed, given the existence of algorithms like the Bakery Algorithm that allow for mutual exclusion amongst an arbitrarily large number of threads, it is

[7]Another, seemingly more efficient, way of achieving this effect is set out on page 3.5.1.

not hard to see that every SVL program is equivalent to one that has no `atomic` constructs at all, although some re-alignment would be needed if such a program were to be put in parallel with other thread(s) that did use `atomic`.

Exercise 3.3.3 Transform the following assignments into atomic-equivalent form.

(i) `a[a[a[0]]] := a[a[1]]`

(ii) `a[y+z] := a[z+y] + a[y+z]`

Exercise 3.3.4 Adapt the code of the functions `IExpEval`, `ffetchi` and `subsi` from `svacomp.csp` to create a program which converts an assignment `Iassign(lv,e)` (where `lv` is a local variable) into an atomic equivalent program.

Now do the same for when `lv` is replaced by an arbitrary expression representing an integer location.

Test your program on the example `a[x+a[x+1]] := y + x*y` and on the assignments in the previous exercises.

3.4 Overseers: modelling complex data types

SVA and SVL only give direct support to very simple data types, namely Booleans, integers, and arrays of these. There is no reason why one should not wish to study programs that share access to more complex objects such as sets, relations, queues, stacks and trees. We might well expect that the operations that nodes carry out on these structures are atomic, and there might well be some internal properties of the representations (data type invariants) that need to be maintained, in the sense that each operation has to maintain it.

For example, one might want to represent a queue or a stack as an array with operations *empty* (represented as a Boolean that is true when the structure is empty), *add* and *remove* (each represented as an integer variable and a Boolean flag that the thread sets when it wants the corresponding operation to happen). Note that both these last two operations are partial: a *remove* is impossible from empty and *add* is impossible when already full.

A convenient way to model these things is via a special sort of process that we call an *overseer*: such a process, while respecting the atomic sections of other threads, always runs with priority every time one of its variables is written. We might then implement the circular array model of a queue as the following thread, where the variables have their obvious meanings:

```
local a:array[0..N-1] of integer;
if add_flag then
   if size=N then signal(overfull)
```

```
            else {size := size+1;
                  last := (last + 1)%N
                  a[last] := add_data;
                  if size == 1 then remove_data := add_data;
                  add_flag := false;
                  empty := false}
else if remove_flag then
    if size=0 then signal(removefromempty)
             else {size := size-1;
                   first := (first+1)%N;
                   if size > 0 then remove_data := a[first];
                   remove_flag := false}
else skip
```

The other threads can access `size` and `empty` like any other locations, but must access the others in the following ways:

```
atomic {add_data := v; add_flag := true}

atomic {v := remove_data; remove_flag := true}
```

Unless we want to add extra and probably unintended atomicity into the given thread, the variable `v` here should be local to the given thread (i.e. not itself shared). So to add `b*c` to the stack, or record the value of a `remove` into `b` where `b` and `c` are shared, we would write

```
v := b*c;
atomic{add := v; add_flag := true};

atomic {v := remove; remove_flag := true};
b := v;
```

These ensure that the accesses to the stack and the other shared variables occur distinctly, rather than atomically. This is, of course, based on the pronciples seen in Section **??**.

Overseer processes do not themselves have atomic sections: they do not need them since they always run with priority. In order to avoid clashes of priority, two overseer processes may not share locations with each other. Overseer processes give themselves priority access to their own locations as follows, as implemented in `svacomp.csp`. An overseer process has four modes,

- `Inactive` (which they are initially). In this mode they allow any process to read and write their locations.
- `Inactive_at (j)`: ordinary thread `j` is in an atomic section, so only it can read and write the overseer's locations.
- `Active_at(j)` is the same except that while in the atomic section, thread `j` has written to at least one of the overseer's locations, meaning that the overseer will go into active mode when the atomic section ends.
- `Active` is entered when either an ordinary thread writes to an overseer variable from a non-atomic position, or when the `Active_at(j)` mode ends. In this mode the overseer runs its own code and allows no other process to read or write its locations. It goes back to `Inactive` when the code terminates.

Now that we have built a queue in this way, it can be used by a single thread, one adding and one removing, or many threads. The processes pushing things onto it will share `add` and `add_flag` with it, and those taking things from it will share `remove` and `remove_flag`. All can read `size` and `empty`. One could have many separate queues (with separately named variables) in a network, with each individual queue action being effectively atomic thanks to the use of overseer processes.

We will see another use of these processes, also involving data-types, in the next section.

Exercise 3.4.1 Modify the queue implementation given above into a (last in first out) stack, in which the bottom end of the stack is always placed at `a[0]`, with the top varying depending on how many items there are. You should create a shared variable to indicate the current size, and partial operations `pop` (analagous to `remove`) and `push` (analogous to `add`).

Exercise 3.4.2 Model a set of integers as a sorted list, with variable `size` and operations `add`, `remove` and `isin`, each of which have a single integer parameter, with the last returning a Boolean that is true if and only if the parameter belongs to the set.

Using the sorted list representation should make `remove` and `isin` efficient.

3.5 Abstraction in SVA

On page **??** we highlighted two problems that mean it is not trivial to verify the whole Bakery Algorithm, namely the unboundedness of the number of threads on the one hand and the ticket values on the other. Each of these can be solved by adding a degree of abstraction into our model.

3.5.1 Two threads out of many

Reading Lamport's original justification of the algorithm in [?], we realise that he concentrates on just a pair of threads. Instead of looking at the global state, he demonstrates that an arbitrary pair of them never break mutual exclusion without relying on what the other threads are doing. Clearly this is a great simplifying insight, since if a given implementation does break mutual exclusion the actions that do breach it come from just two processes.

Concentrating on the original algorithm, we can build an implementation in which the mutual exclusion of a particular pair of threads is examined by giving only these two the ability to perform the signals `css.i` and `cse.i`.

We can assume that the two threads we are considering are `i < j`. The only effects the other threads `k` now have is in the values they set for the shared variables that are read by threads `i` and `j`.

For specific values of `i` and `j` it makes sense to try to prove that these two threads are mutually exclusive by running the actual definitions of these threads in a context where the variables set by other threads (`choosing(k)` and `number(k)` for `k!=i,j`) are always arbitrary. For if threads `i` and `j` achieve mutual exclusion in that context they will achieve it in parallel with any other threads at all that are subject to the same rights to read and write shared variables.

The author has implemented these arbitrary values in two different ways. The first was to run simple processes as the "other" threads whose only activity is to repeatedly assign random values (as for on page 89) to the locations associated with them. The second was to run only threads `i` and `j`, and to put all of the other threads' locations in `ExtWrites` (see page ??). This means that these locations can now have an arbitrary value assigned to them at any time by an write from "thread" `-1`. This proved enormously more efficient for all the variations on the Bakery Algorithm that we have considered (original and simplified, with many variations on the *maximum* algorithm, and with and without dirty variables). The results (see `bakeryproof1.csp`) were almost exactly as expected, but were slightly tidier in respect of the incorrect maximum algorithm `Maximum D`. The parameters used here were `MaxI=10`, `N=5`, `i=2` and `j=4`.

- Without dirty variables, both original and simplified algorithms satisfy mutual exclusion for versions `A, B, C` of `Maximum`, and fail for all four versions of `Maximum D` (see page ??).
- Exactly the same result holds for the original algorithm with dirty variables, but all versions of the simplified algorithm then fail.

This choice of the parameters other than `MaxI` is clearly a significant one, since it means that there is another node less than both `i` and `j`, one between then and one greater than both. A little thought reveals that this demonstrates the correctness of the algorithm for any number of threads, since the effects that an arbitrary number of threads in any one of these three gaps can have are no more general than the single randomising thread. We can, however, make this clearer by making threads 2 and 4 look at notional threads in these gaps a nondeterministic number of times, so that any behaviour possible of any such pair is contained in our abstracted system. Thus, in both the maximisation and waiting loops, we make these nodes wait for threads `1,3,5` a nondeterministic number of times (including zero). The following shows a version of `Maximum B` modified to have this effect. The uses of `(k mod 2)=0` just exploit the fact that our two real threads have indices 2 and 4, and the notional ones have indices 1,3 and 5. Here, `randb` is a random Boolean.

```
k := 0;
while k<=N do
 {if (k mod 2)=0 or randb then
          number(i) := max(number(k),number(j))
  if (k mod 2)=0 or randb then k := k+1};
number(i) := number(i)+1;
```

The file `bakeryproof2.sva` shows this applied to maximum `A` and `B` in the original algorithm. These are, of course, correct with or without dirty variables.

We have therefore given a complete automated proof that these versions of the Bakery Algorithm work for any number of threads at all. The strategy of this proof was:

1. Break down the correctness condition you want to prove into many similar parts, parameterised by the threads that actually perform the signal events seen in a counter-example trace for the overall specification. Here there are two threads: the one `n` already in its critical section, and the one `m` that enters its illegally at the same time.
2. For each such part, identify the (hopefully) small number of threads that we rely on to make that part true. In our case these are just `n` and `m`.
3. Abstract away the other threads and their actions on the variables that threads `n` and `m` read in an appropriate way.
4. Abstract away the relevant threads' (for us `n` and `m`) interactions with the others in a way that simulates the ways these behave with any number of others present.

In some examples these may not always be possible, but even the factorisation of the original check into parts might give benefits thanks to the increased amount of compression possible when some signal events are hidden.

EXERCISE 3.5.1 Consider the following assertion about the original Bakery Algorithm: *The value assigned by the line*

$$number[i] := 1 + maximum\{number[1], \ldots, number[N]\} \quad (*)$$

is, for any j, at least $1 + min$, where min is the least of the initial value of number(j) and all values assigned to it during the period when $(*)$ is being executed.

Formulate a file that seeks to prove this for general i and j, emulating the structures in this section. [Note that the cases $i < j$ and $j < i$ are now distinct since the specification is not symmetric in i and j.]

Is this property true for the following cases?

(i) `Maximum A` with clean variables.

(ii) `Maximum D` with clean variables.

(iii) `Maximum A` with dirty variables.

3.5.2 Finitely representing an infinite linear order

Every check we have carried out to date on the Bakery Algorithm has been with a sub-range of the non-negative integers as ticket numbers. We have had to ignore the run-time errors that appear when the `number[i]` or `turn[i]` values exceed this limit. We cannot therefore claim to have proved that the Bakery Algorithm always works, only that no error can appear before these values get out of range.

As was the case with the number of threads, we can solve this problem by generalising, or abstracting, the system. In the discussion below we will concentrate on the version in which we are examining two threads programmed to simulate an arbitrary number in the three gaps in the thread ordering, but exactly the same strategy can be used with any fixed number of threads.

Just as in the previous case, we need to find a way of simulating every behaviour of the algorithm when it is using the integers stored in `number` with how it behaves when this is replaced by a well-chosen finite structure. The key to discovering this is to examine what threads do to these integers:

- They compare integers, both in the maximisation process and when waiting for another thread to complete its critical section.
- They add one to an integer, where the role of this operation is clearly to find a value that is strictly larger than the one that is added to.

- Constants with known position in the order are assigned to array components, as are the results of additions and maximisations. (There is one, namely `0` in the original Bakery Algorithm. The simplified algorithm also uses `1`.)

We can regard these integers and the ones that represent loop/thread indexes as separate types. We will call the type of `number[i]` values *Ticket*. In fact the program would work with *Ticket* any linear order L (i.e. a transitive relation $<$ such that exactly one of $x < y$, $x = y$ and $y < x$ holds for all x and y) with the following properties:

- We have a fixed element 0 of L.
- For every element x of L we can find $next(L)$ which is the least element strictly greater than x.

It is clear that we can implement *max* over any linear order, and $next(x)$ takes the place of $x + 1$. What we will do in this section is to show how to build a finite model that establishes the correctness of the Bakery Algorithm implemented over any such order.

It is certain that we cannot use such a type *Ticket* directly in our implementation, since the model would them be anything but finite. What we will in fact do is produce a finite model using a special finite *FinTicket* type and which can simulate any behaviour for an arbitrary *Ticket*. In other words, any execution of the Bakery Algorithm implemented over *Ticket* has an analogue – with the same `css.k` and `cse.k` actions, as the original.

The way we will do this not by making *FinTicket* a finite linear order – for `next` cannot be universally defined – but by making it record the relative place the location it represents has amongst the *occupied* members of *Ticket*, namely the values of *Ticket* that are held by some location (whether shared or local) meaningful to at least one thread.

There are one or two special values in the type *Ticket* depending on whether we are using the original or simplified algorithm, namely 0 and perhaps $1 = next(0)$. We need to record accurately whether a specific location (a *slot*) has one of these values. We will therefore, for the original algorithm where only 0 is special imagine a map from the *Ticket* values in the state of our system to $0 < v_1 < ... < v_M$ (where M is greater than the total number of such values) as follows:

- Any value which is 0 will map to itself.
- Any value which greater than 0 will map to v_k, where that value is the kth smallest amongst all those values that are not 0.

If we do this then we can still decide every comparison between two ticket values, since this mapping is always stictly monotonic. For the system we are considering, of two real threads reading their own locations and those of three notional ones, the transformation clearly achieves the goal of making it finite state. However it also creates some problems:

- If a single ticket value is changed, we might have to change the index of every single slot, for example because a slot that was previously the only one having value 1 is given either 0 or becomes the maximum one.
  The solution to this problem is to maintain, using an overseer, the indexes using a monitor that refreshes them completely every time one is changed.
- This in turn creates a problem when a thread is in the middle of a calculation involving the indexes, since the overseer will not alter the values that our thread may have input but not yet used. So, for example, if the overseer changes the indices x and y while the thread is carrying out `x := max(x,y)`, the value that gets assigned may be the greater of the *old* value of `x` and the *new* value of `y`.
  The solution to this is to make all of our threads' operations atomic. In general, of course, this would change their behaviour; but we have already shown how to build atomic equivalent programs where this is not an issue.
- If *next* is applied to a value that is not already the greatest, we *do not know* whether it should become equal to the next greatest value or get a new value beneath that. For the answer depends on what the actual ticket values are, not their indices which is all that we are remembering.
  Our solution is to make a nondeterministic choice whenever this situation arises. As we will see later, this can require quite a bit of shuffling of the indices, particularly as we may be creating an extra member of the order in the middle.
  It is important to realise that by converting what is in fact a deterministic choice into a nondeterministic one, here on the grounds that we have abstracted away the data that is required, we are creating an anti-refinement or abstraction of the original behaviour, just as we did in the previous section when assuming that "other" threads could assign anything. A counter-example in an abstracted system of this sort does not guarantee that there is one in the original, but a proof that the abstracted system refines some specification is valid for the original.

We can therefore use the ideas of overseers, atomic equivalents and abstraction, all seen in previous sections, to complete our proof.

The following is program for the original Bakery Algorithm with `Maximum A`, modified so that (i) it takes account of the proof methods for arbitrary numbers of threads seen in the last section and (ii) every use of an identifier of *Ticket* type can be made atomic. with `Maximum A`.

```
iter
  {choosing[i] := true;
   temp := 0;
   j:= 1;
   while j< N do
    {if randb or (j mod 2 = 0) then
                temp := max(temp,number(i));
     if randb or (j mod 2 = 0) then
                j := j+1
    };
   number(i) := temp+1;
   choosing[i]:=false;
   j := 1;
   while j< N do
    {
     if randb or (j mod 2 = 0) then
       {while choosing[j] do skip;
        lnj := number[j];
        b := lnj>0 and ((lnj,j)<(number(i),i));
        while b do
          {lnj := number[j];
           b := lnj>0 and ((lnj,j)<(number(i),i));
          };
        };
     if randb or (j mod 2 = 0) then
       j := j+1
     }
   signal(css.i);
   signal(cse.i);
   number[i] := 0
  }
```

To run, the abbreviations `(lnj,j)<(number(i),i)` are expanded into `(lnj < number(i)) or ((lnj = numbe` which makes no difference in evaluation since all these variables are local or (in the case of `number(i)`) writable only by this thread. Note that `Maximum A` required no transformation to make its assignments equivalent to atomic ones, but that a significant transformation was required to the second half of the program.

Now we have this code we can enumerate the slots whose positions need to be remembered in the order. The obvious inteference is that there are `N=5` slots `number(k)` and two local ones for each of our two active threads, namely `temp`, `lvj`. In fact we can do better than this:

- We do not need to reserve a slot in the partial order for the locations `number(1)`, `number(3)` and `number(5)`: the only role of these is to read to random member of the order into the threads (2 and 4) that we are modelling properly. What we do need to ensure is that when these are read they might give values in any place in the order of the read slots.
- Examining the programs for threads `2,4` reveals that `temp` and `lvj` are used in separate phases of the program: we can use a single slot of each: one for each of threads `2` and `4`. We will call these threads `temps(2)` and `temps(4)` below.

It follows that we need the following room in our order:

- The special value `0` that needs to be kept separate because it has a definite control effect on the program.
- Room for the four slots mentioned above, all of which may be different and different from `0`.
- Both through reading the random `number(i)` and the incrementing step at the end of the initial maximisation, a value can be calculated that is not one of the present four or `0`. We therefore include extra spaces in the order (one between each consecutive pair, and an extra one at the top) for this purpose.

We therefore use `{0..9}` as i*FinTicket*, interpeting `0` as itself, mapping the four slots that are not `0` to as many of `2,4,6,8` as are needed, with the odd numbers playing the roles of the gaps. The overseer's duty will be place our fours slots in order like this whenever one of them is changed.

So, for example, if

```
0 = number(2) < number(4) = temps(4) < temps(2)
```

then the overseer will map `number(2)` to `0`, `number(4)` and `temps(4)` to `2`, and `temps(2)` to `4`. The assignment `number(4) := temps(4)+1` will certainly lead to `number(4)` being greater than `temps(4)`, but from the information above we do not know whether it is still less than, or is equal to `temps(2)`. This reflects the nondeterminism discussed above. The way we implement this is to modify the assignment so that it randomly adds `1` or `2` except when the result would be `10`.

This therefore becomes a nondeterministic operation on our abstracted type rather than a deterministic one on the integers.

The reason why we need this nondeterminism in this example is that we cannot tell from the above whether `temps(2)` is only `1`, or at least `2`, greater than `temps(4)`.

What the overseer produces is a standardised representation of the ordering of `0` and the four slot values. The values used are independent, for example, of adding any constant onto the non-`0` slots. We therefore have a finite-state representation of an arbitrary mapping of the slots into a suitable linear order: so we have collapsed the infiinity of tuples of slots that can arise in runs with any suitable order such as the non-negative integers into a finite number, which is in fact 132 (1 in which all four slots are `0`, 4 in which three are, 18 in which two are, 52 in which one is, and 57 in which none are `0`.

One definition of an overseer process that achieves this is given below. Here `hits` is a Boolean array initialised to `False` that is used to record which of the values `{0..9}` is present in the four slots. These are then used to construct a mapping `omap` from this set to `{0,2,4,6,8}` based on the principles discussed above, and the mapping is applied to the four slots to standardise their values.

```
hits(number(2)) := True;
hits(number(4)) := True;
hits(temps(2)) := True;
hits(temps(4)) := True;
nextw := 2;
kk := 0;
while kk<9 do
  {kk := kk+1);
   if hits(kk) then
     {omap(kk) := nextw;
      nextw := nextw+2};
   hits(kk) := False}
number(2) := omap(number(2));
number(4) := omap(number(4));
temps(2) := omap(temps(2));
temps(4) := omap(temps(4));
kk := 0;
while kk<9 do
   {kk := kk+1),
    omap(kk) := 0}
```

Notice how the loops here are structured so that the integers `kk` and `nextw` never exceed `MaxI=9`. The combination threads `2` and `4` and this overseer are implemented in `bakeryproof2.svl`. This is a good-sized check (over 250M states after two compressions involving processes with more than 1.1M states), in which the author exprimented with different allocations of locations to these three and some "dummy" threads to produce a reasonably efficient result. See that file for discussion of this.

The fact that this system passes the mutual exclusion specification *and has no run-time errors* means that we have proved that the original Bakery Algorithm with version A of *maximum* guarantees mutual exclusion for any number of threads provided that the *Ticket* values are chosen from a linear order of the sort described on page **??**, with the `+1` interpreted as the *next* operation. For we have shown that the control behaviour of our thread models (where ticket values are drawn from the abstraction of an order) is an anti-refinement of that of the ones defined using the real order, where in each case ticket values from nodes other than these two are treated as random. We saw in the previous section how the proof for this arbitrary pair establishes correctness for an arbitrary number of threads.

We should reflect on the use of atomic equivalent form in this proof. It is necessary because of the particularly intrusive nature of the overseer process we have used here. The examples seen in Section **??** all maintained the internal consistency built using integer and Boolean values that were interpreted as such in the world outside the overseers. The role of the overseer in the present section is to maintain a partial injective mapping from the infinite type used by the real-world threads to the finite type used in our representation. The representation of a single value can change *while a thread process is holding it.* In the same example qoted above:

```
0 = number(2) < number(4) = temps(4) < temps(2)
```

if the assignment `number(2) := 0` occurs then, while the real-world values of the other three slots do not change, their representations all decrease by `2` (from `4` and `6`). The overseer changes these values as they sit in the variables, so all is well provided no thread is either (i) partway through evaluating an expression based on several values altered by the overseer or (ii) has calculated a term based on pre-change values but not yet output this value (either by writing to a location or along an `isignal`). These bad cases are excluded by running the atomic equivalent form with all assignments made atomic.

So what the main check in `bakeryproof2.svl` actually does is prove that the atomic equivalent of the stated version of the algorithm works. We can infer that the ordinary version of this algorithm works because we know that any thread is equivalent to its atomic equivalent.

The proof presented in this section is complex, but it does go well beyond the traditional role of model checking, namely the proof of a fixed and finite-state version of a system. It was achieved through two different sorts of abstraction, the first being to replace thread behaviour that depends on some parameter (in this case the number of threads) by more nondeterministic behaviour that does not. The second was to discover what properties of an infinite data type our threads appeared to be using (in our case the relative order of ticket values) and find a way of getting the same information from a finite representation.

It is straightforward to adapt the proof method set out in this section to other variants of the Bakery Algorithm, bearing in mind that this exercise only makes sense if the simpler transformation presented in Section **??** meets the mutual exclusion specification with error events hidden. The same technique also works for the final version of Simpson's algorithm on page **??** with the counter. Since, in that example, one process generates the values of the counter in sequence and all the other one does is read, copy and compare them, the linear order there can be mapped to consecutive integers rather than leaving the odd numbers as gaps.

It seems likely that the same approach will work wherever an otherwise finite-state system depends on counter-like variables where it is the relative order of these rather than the actual values (aside from a finite set of special values like `{0}`).

This technique will also work with other infinite data structures such as partial orders and graphs, where abstractions suitable for a given application can be found.

EXERCISE 3.5.2 Adapt the proof in this section so that it works for `Maximum B` and `Maximum C`. Do these proofs require four slots, more or less?

EXERCISE 3.5.3 Adapt the proof in this section to the *simplified* Bakery Algorithm. How many slots are required?

EXERCISE 3.5.4 Prove the sequential consistency and recentness of the version of Simpson's algorithm from page **??** with counters, so that the effect of an unbounded space of counters is reproduced in a finite type as discussed above.

3.6 Notes

Simpson's four slot algorithm [**?**] was brought to my attention in September 2008 by John Fitzgerald. Much of the analysis of it in Section **??** parallels that in John Rushby's paper [**?**], though using rather different tools, and the idea of write-when-different comes from that source. The augmentation with counters is, as far as I am aware, original.

The first implementation of overseers in SVA was by David Hopkins [**?**]. These were introduced to avoid difficulties that he and I had discovered in other models of prioritised execution in SVL-style processes. In those, we had only allowed low-priority threads to perform a step when high-priority ones specifically executed an `idle` statement. Since typically this led to high-priority threads evaluating an expression to decide if each `idle` is allowed, it gave rise to some very unwieldy results. Since none of the activities of low-priority processes could be hidden when they have to synchronise with `idle`, this type of prioritisation would have greatly damaged the effects of compression. For these reasons the present version of SVA does not include this latter priority model, relying rather on overseer processes that do nothing until particular locations are written to.